Technische Universität Carolo-Wilhelmina zu Braunschweig

Department of Computer Science

Master's Thesis

Assessing Performance Evolution Of Configurable Software Systems

Untersuchungen zur Performanz-Evolution von konfigurierbaren Software-Systemen

Author:

Stefan Mühlbauer

Matriculation Number: 4161884

December 18, 2017

Advisors:

Prof. Dr.-Ing. Ina Schaefer

Technische Universität Braunschweig · Institute for Software Engineering and Automotive Informatics

Prof. Dr.-Ing. Norbert Siegmund

Bauhaus University Weimar · Chair for Intelligent Software Systems

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Masterarbeit "Assessing Performance Evolution for Configurable Software Systems" selbstständig verfasst, alle benutzten Quellen und Hilfsmittel vollständig angegeben habe und die Arbeit nicht bereits als Prüfungsarbeit vorgelegen hat.

Braunschweig, December 18, 2017 _____

Abstract

Most modern software systems are, to varying extents, configurable. Configurations influence the performance behavior of configurable software systems, and, therefore, can impact the software system's quality. Even small numbers of configuration options (features) provided can result in an infeasible number of configurations due to combinatorics. Moreover, some effects only arise when multiple configurations are selected (feature interaction). To tame this complexity due to a software system's variability, approaches emerged to learn and estimate a configuration's effect on performance measurements (performance prediction models). While performance of a configurable software system can be relatively well understood with few sample measurements for a single version, as software evolves, performance evolves for each variant as well. Postponed quality assurance and maintenance tasks in general, but in particular the exhibited complexity of configurable software systems, pose a risk to a software systems quality as those evolve; this technical debt can eventually result in performance degradation. So far, there exists no means to assess the performance evolution history of configurable software systems.

With this we thesis provide a methodology about how to unveil the performance evolution history of configurable systems: We provide a guideline to untangle a software systems variability, a catalog of strategies to select versions for which performance evolves significantly, and a guideline about how to measure and summarize performance, in particular in the context of variability. In a case study, we evaluate our methodology for two configurable software systems. We document our findings, and, using the case study results, we obtain valuable insights about the quality of the software systems for which performance evolution was assessed.

Zusammenfassung

Moderne Softwaresysteme bieten oftmals eine Vielzahl an Möglichkeiten zur Konfiguration. Jede einzelne Konfiguration kann sich wiederum auf die Performanz der Software auswirken, also die Leistungsfähigkeit verbessern oder verschlechtern. Jedoch treten einige Effekte nicht für einzelne Optionen, sondern auch für mehrfache Kombinationen jener auf. Diese Interaktionen zwischen Optionen stellen einen wesentlichen Grund für die Komplexität von konfigurierbaren Softwaresystemen dar. Zwar lässt sich die Performanz einer Variante eines konfigierbaren Softwaresystems mittlerweile gut abschätzen, jedoch kann sich die Performanz einer Variante unabhängig von anderen entwickeln. Unzureichende oder verzögerte Wartung sowie Qualitätssicherung bieten hier ein Einfallstor für eine sich über die Zeit verschlechternde Architektur und somit letztendlich möglicherweise Einbußen in der Performanz. Bisher existiert kein Verfahren, um die Evolution von konfigurierbaren Softwaresystemen unter Gesichtspunkten der Performanz zu untersuchen.
Mit dieser Arbeit schließen wir diese Lücke und entwerfen eine Methodologie um die Evolution konfigurierbarer Systeme unter Performanzaspekten nachzuvollziehen und besser zu verstehen. Unsere Methodologie umfasst Leitfäden zur Analyse der einem System zugrundeliegenden Variabilität, Strategien um solche Versionen auszuwählen, für welche sich die Performanz signifikant ändert. Darüber hinaus umfasst sie eine Zusammenstellung von Werkzeugen um Performanzmessungen durchzuführen, zusammenzufassen und zu analy-sieren. Wir evaluieren unsere Methodologie mit einer Fallstudie zu zwei konfigurierbaren Systemen. Aus den Ergebnissen gewinnen wir wertvolle Rückschlüsse über die Qualität der betrachteten Softwaresysteme.

Contents

List of Figures

List of Tables

1 Introduction

Configurable Software Systems. Modern software systems often need to be customized to satisfy user requirements. Configurable software, for instance, enables greater flexibility in supporting varying hardware platforms or tweaking system performance. To make software systems configurable and customizable, they exhibit a variety of *configuration options*, also called *features* (Apel et al., 2013). Configuration options range from fine-grained options that tune small functional- and non-functional properties to those that enable or disable entire parts of the software system. The selection of configuration options can be accommodated at different stages, either at *compile-* or *build-time* when the software is built, or at *load-time* before the software is actually used. Compile-time variability usually governs what code sections get compiled in the program. Runtime-variability allows to configure the system during execution, which is needed, for example, for context-sensitive systems. For instance, compile-time variability can be realized by excluding code sections from compilation using preprocessor annotations (Hunsen et al., 2016) or by assembling the code sections to compile incrementally from predefined code modules depending on the feature selection (Schaefer et al., 2010). In contrast to that, load-time variability controls which code sections can be visited during execution. Configurations for load-time variability can be specified using configuration files, environment variables, or command-line arguments. Many software systems are configurable, examples range from small open-source command-line tools to mature ecosystems including Eclipse or even operating systems, such as the Linux kernel with more than 11,000 options (Dietrich et al., 2012).
Configuration options for software systems are usually constrained (e.g., are mutually exclusive, imply or depend on other features) to a certain extent. In the worst case though, at which all options can be selected independently, the number of valid configurations grows exponentially with every feature added, and exceeds the number of atoms in the entire universe once we count 265 independent features. Hence, even for a small number of features, any naive approach for assessing emergent properties of configurable software systems exhaustively for each valid configuration is conceived infeasible. Despite this mathematical limitation, many feasible approaches to static analysis for configurable systems emerged. Those variability-aware approaches enable, for instance, type checking in the presence of variability by exploiting commonalities among different variants (Thüm et al., 2014).
To meet functional and non-functional requirements, users aim at finding the optimal configuration of a configurable software system. However, this task is non-trivial and has shown to be NP-hard (White et al., 2009). The main driver for the complexity are feature interactions. A *feature interaction* is an "emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved" (Apel et al., 2013) and can make development and maintenance of a configurable system an error-prone task. To illustrate feature interactions, consider the following example (Siegmund et al., 2015): A software system, say a file server, is used to

store files in a database and provide access upon request. The system provides two features, encryption and compression. In isolation, both file en- or decryption and file (de-)compression demand an expectable fraction of memory and processor time. The performance behavior for the software system though may vary if both features are selected. For instance, if a file is encrypted and compressed (or vice versa), we can expect the operation to demand less resources since an compressed file is likely to be of smaller size than the decompressed original. This is a positive example for a feature interaction, where the performance behavior, although being benefiting, is unexpected.

Performance Behavior. The term "performance" with respect to software and software systems is not precisely defined and differs from an end user's and developer's perspective. According to Molyneaux (2014), from a user's perspective "a well-performing application is one that lets the end user carry out a given task without any undue perceived delay or irritation". However, to accurately assess performance, from a practitioner's perspective, performance is outlined by measurements called *key performance indicators* (KPIs) which relate to non-functional requirements (Molyneaux, 2014). The set of KPIs includes availability of a software system, its response time, throughput, and resource utilization. Availability comprises the amount of time an application is available to the user. Response time describes the amount of time it takes to process a task. Throughput describes the program load or number of items passed to a process. Resource utilization describes the used quota of resources required for processing a task. The performance behavior of a software system depends on the functionality offered, the respective implementation, program load, the underlying hardware system, environment variables, and the resulting execution. Since configuration options control what and how functionality is executed, we concentrate here on this source of performance. While feature interactions not necessarily cause the software system to break severely in all cases, its overall performance can become unfavorable for corner cases or specific configurations as the feature selection influences the execution. That is, the choice of features influences the performance of a software system.

Performance and Evolving Software. Actively maintained software systems evolve with every modification made, every version released, and patch provided. Modifications usually introduce new functionality to the system, but functionality might as well be divided into smaller modules to provide more fine-grained configuration options. When features are removed from the software system, the corresponding functionality might remain in the code base or options are merged (Apel et al., 2013).
There exists substantial work on understanding the evolution of configurable systems, for instance, with respect to software architecture (Zhang et al., 2013; Passos et al., 2015) or variability (Seidl et al., 2012; Peng et al., 2011; Passos et al., 2012). As software evolves, the code base which is subject to modifications and the overall architectural quality can degrade. Common symptoms of architectural degradation are code tangling and scattering (Zhang et al., 2013; Passos et al., 2015), which lead to less cohesive and stricter coupled code. The more the code base is constrained and interdependent, the more software can become "brittle" (Perry and Wolf, 1991), less flexible, harder to adapt, and therefore harder to evolve. Evolution of software, especially with respect to variability, is essentially driven by and can be conceived as adapting a software system to changed requirements and contextual changes (Peng et al., 2011). That is,

(potential) degradation of software quality as software evolves is often a phenomenon due to decisions trading quality assurance (QA) and maintenance tasks with meeting requirements and schedules (Guo et al., 2011). The metaphor of *technical debt* (Guo et al., 2011) which is commonly used to describe this trade-offs and corresponding costs, outlines the risk that postponed maintenance tasks pose to software evolution. Although every deferred maintenance or QA task may save some cost, it also could have unveiled software defects in the first place. Technical debt implies both interest, so to speak, the potential damage of a defect depending on its severity, as well as the probability of incurring interest. A defect can be severe, yet fixable with reasonable effort and cost. However, the aforementioned symptoms of architectural degradation and deferring maintenance render bug-fixing to become more and more expensive.
Besides the aspects of software evolution discussed above, the evolution of performance for software systems has gained more attention recently. In practice though, quality assurance with respect to performance is still conducted to an unsatisfactory extent, or accommodated too late in the development process, according to Molyneaux (2014). Thus, postponed maintenance and QA with respect to performance is likely to a driving factor for degradation of performance quality, or simply called *performance regression*. While performance engineering has emerged as a discipline of or target in software testing, qualitative root-cause analysis, for the most part, is still conducted manually (Molyneaux, 2014). However, there exists work on automated root-cause analysis for performance bugs, such as measuring the execution time of unit tests, whereby an increased execution time indicates performance regression, and the corresponding unit test helps isolating the root-cause thereof (Heger et al., 2013; Nguyen et al., 2014b). In conclusion, we see that, to better assure good software performance, more knowledge about performance behavior needs to be available, ideally, earlier in the development process.

Performance Prediction. For configurable software systems, performance behavior can be more complex and dependent on the feature selection, as we have seen with the example for feature interactions above. Similarly, quality assurance for configurable software systems is far from exhaustively testing all possible configurations, but rather close to only testing a selection of configurations sampled with respect to certain constraints. Sampling strategies might stress feature interactions, such as pairwise sampling (Siegmund et al., 2012). However, all samples are selected with the intention to learn as much as possible about the entire system from a small sample set of variants. So to speak, a sampling strategy is "optimal" if for a resulting sample set, the probability of missing an arbitrary (relevant) feature interaction, is minimal.
While performance testing is apparently useful, recently, a number of techniques to model and predict performance behavior for arbitrary configurations have emerged (Siegmund et al., 2012, 2015; Guo et al., 2013; Sarkar et al., 2015). The underlying optimization problem of performance-prediction models is to find an accurate estimator $\hat{f}$ for a function f describing a performance property depending on the feature selection. Performance properties can be estimated without performance measurements, for instance by inferring performance properties from software models (Woodside et al., 2007); measurement-based approaches for configurable systems address this optimization problem. The proposed approaches include learning performance behavior with decision trees (Guo et al., 2013; Sarkar et al., 2015), learning a frequency-based repre-

sentation of the target function (Zhang et al., 2015), or learning the influence of single features and all performance-relevant feature interactions (Siegmund et al., 2012, 2015). All approaches have shown promising error rates for several real-world applications and allow prediction of system performance for arbitrary configurations. To create performance-prediction models, all approaches demand samples of performance measurements for multiple configurations.

1.1 Problem Statement

The assessment of performance evolution requires a series of performance-prediction models describing performance behavior for a series of versions of a corresponding configurable software system. Assessing the performance behavior for a single version of a configurable software system entails a number of necessary and preliminary tasks. However, these tasks become even more complicated once a series of versions needs to be assessed:

- *Variability Model Synthesis:* Not all configurable software systems do explicitly exhibit a variability model which is, however, required to derive all valid variants (Rabkin and Katz, 2011; Nadi et al., 2015). While substantial work exists on reverse engineering variability models from code (Rabkin and Katz, 2011; She et al., 2011; Zhou et al., 2015; Nadi et al., 2015) or non-code artifacts (Alves et al., 2008; Andersen et al., 2012; Bakar et al., 2015), many techniques still involve manual decisions (She et al., 2011) and domain knowledge (Nadi et al., 2015). Moreover, variability models evolve as part of the software (Peng et al., 2011), vary from version to version, and therefore, require repeated reverse engineering steps.

- *Version Sampling:* To study performance evolution, we need to specify which snapshots or versions of a software system are relevant to describe its performance behavior over time. While detecting releases and release candidates should be straightforward, one might, for instance, be interested in the performance evolution including snapshots between two releases, for example after bug fixes. Since it is often the case that not all snapshots do compile, classifying defect snapshots can still be tedious work.

- *Performance Assessment:* The accurate assessment of performance evolution requires a suitable testing setup. The methodology required for assessing performance among others requires the selection of suitable performance metrics and corresponding benchmarks, means to record measurements, and repeat experiments easily as well as proper ways to interpret and compare results.

Goals and Thesis Structure. The goal of this thesis is to provide a theoretical and practical methodology to enable exhaustive performance measurements of configurable software systems and over their version history. That is, we contribute a guideline of and tool support for performance measurements of configurable and evolving software systems. Our research objectives and desired outcomes are

- a literature overview regarding software evolution, feature model synthesis, and performance assessment,

- a methodology to assess performance evolution with respect to the aforementioned challenges, and
- a practical tool for performance measurement for multiple revisions of configurable software systems.

We also propose an approach of automatically detecting promising versions of the configurable software systems for performance measurements and evaluate whether our assumptions hold.

1.2 A Methodology for Assessing Performance Evolution

The methodology described in this thesis is organized with respect to three dimensions of configurable software systems: *variability*, *version history*, and *performance*. The schematic overview of the performance evolution assessment process shown in Figure 1.1 outlines the major aspects of each dimension.

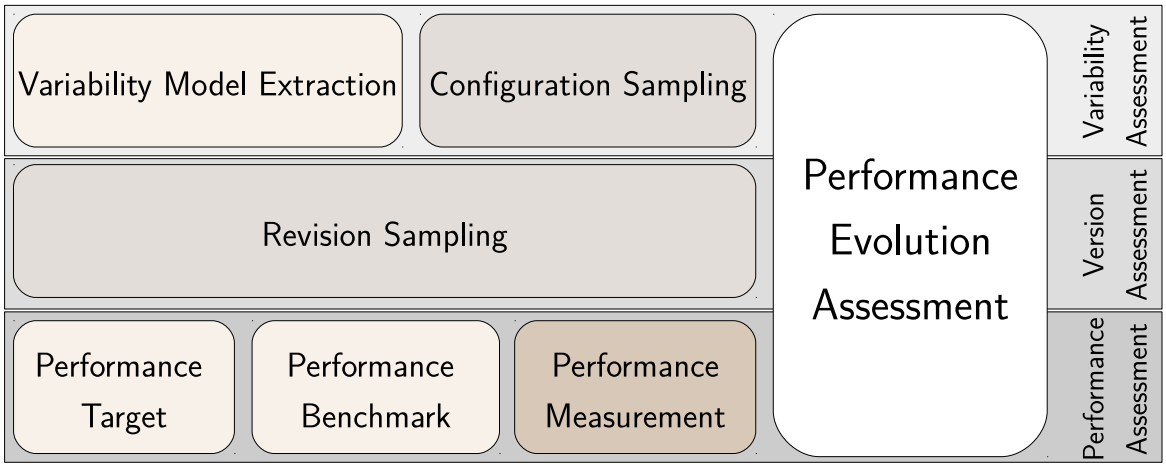


Fig. 1.1. Overview of the the proposed methodology including the necessary processes.

First, objectives related to the variability of a configurable software system are considered. This includes the extraction and comprehension of knowledge about the system's variability. Second, we address objectives which emerge with evolving software, for instance whether a system compiles for a specific version, or whether a version is a promising candidate for detecting performance changes. Finally, we describe in detail the assessment of performance which comprises the selection of suitable performance benchmarks as well as a selection of appropriate statistical means to summarize, compare, and evaluate performance statistics in order to derive meaningful insights.
In addition to the three aforementioned dimensions in Figure 1.1, performance evolution assessment can also be conceived as consisting of three different categories of tasks, as the different colorizations indicate. First, for a configurable software system, its variability needs to be assessed in order to obtain a variability model to derive and

select configurations from. Second, for a software system, a sample set of revisions needs to be selected. Since covering all variants and versions is not feasible, a sampling strategy needs to be chosen which is likely to uncover performance changes. Finally, the performance assessment goals are specified and corresponding measurements are conducted and evaluated as mentioned above.

The thesis is organized as follows. Chapter 2 provides the background to the relevant topics discussed in this thesis, including variability modeling, software evolution, variability model synthesis, performance assessment, statistical basics to summarize data records, and performance prediction models. In addition, we review existing work related to the thesis topic. In chapter 3, we review existing approaches to understanding and comprehending variability in configurable software systems. We categorize and summarize methodological strategies to synthesize variability models and discuss different configuration sampling strategies. In chapter 4, we construct and evaluate methodological strategies to select revisions likely to have an impact on performance from the overall version history to sketch performance evolution efficiently. In chapter 5, we propose our performance measurement methodology along with statistical considerations regarding the summarization and presentation of measurements results. In Chapter 6, we evaluate several aspects of our tool with respect to practicality and discuss the results thereof. Finally, chapter 7 concludes the thesis and gives an outlook on possible future work.

2 Background and Related Work

This chapter presents necessary background to the individual topics used in this thesis. In section 2.1, we recapitulate basic concepts, notations, and terminology regarding variability modeling. In section 2.2, we outline different aspects of software evolution. section 2.3 discusses the assessment and testing of performance. Section 2.4 recalls methodology to predict performance behavior for configurable systems. Finally, in Section 2.5 we review a body of related work that has either touched on topics similar to ours, or pursues a similar methodology, yet with different basic premises.

2.1 Variability Modeling

The design and development of configurable software systems is conceptually divided into *problem space* and *solution space* (Czarnecki and Eisenecker, 2000). The problem space comprises the abstract design of features that are contained in the software system as well as constraints defined among features, such as dependencies or mutual exclusion. The solution space describes the technical realization of features and the functionality described by and associated with features, e.g., implementation and build mechanisms. That is, features cross both spaces since they are mapped to corresponding code artifacts.

A common way to express features and constraints in the problem space is to define a *variability model*, or *feature model*, which subsumes all valid configurations (Kang et al., 1990; Thüm et al., 2009; Apel et al., 2013). There are different and equivalent syntactical approaches to define feature models, for instance, a propositional formula F over the set of features of the configurable software systems (Batory, 2005). In this case, a configuration is valid with respect to the feature model if and only if F holds for all selected features being true and all unselected features being false, respectively. However, a more practical and more commonly used way to express feature models are graphical tree-like *feature diagrams* (Apel et al., 2013). In a feature diagram, features are ordered hierarchically, starting with a root feature and subsequent child features. By definition, the selection of a child feature requires the parent feature to be selected as well. Child features can either be labeled as *optional* features or *mandatory* features; the latter ones need to be selected in every configuration. Moreover, feature diagrams provide a syntax for two different types of feature groups, *or-groups* and *alternative-groups*. For an or-group, at least one of the group's features needs to be selected for a valid configuration, whereas for an alternative-group exactly one out of the group's mutually exclusive features must be selected. In addition to the feature hierarchy, constraints, which cannot be expressed by the tree-like structure, are referred to as *cross-tree constraints*. Cross-tree constraints, depending on the notation, are depicted by arrows between two features or simply added to the feature diagram as a propositional formula. For two features f_1 and f_2, a cross-tree constraint means that for feature f_1 to be selected, either the selection of f_2 is required/implied or excluded.

An introductory example for the syntax and semantics of feature diagrams is provided in Figure 2.1. In this example, an imaginary vehicle propulsion can be configured with eight valid configurations. The vehicle requires an engine, and therefore feature Engine is mandatory. At least one out of the three features Hybrid, Piston and Electric needs to be selected. For a piston engine, we can select either the feature Diesel or Petrol. A petrol engine requires additional ignition sparks in contrast to a Diesel engine. For an electric engine, we require a battery, hence, the feature Battery is mandatory. In addition, the feature model specifies two cross-tree constraints: First, the feature Tank is optional, yet once a piston engine is selected, we require a tank. Second, if we want to use the Hybrid functionality (e.g., use both electric and piston engine simultaneously), we require to have both a piston and an electric engine.

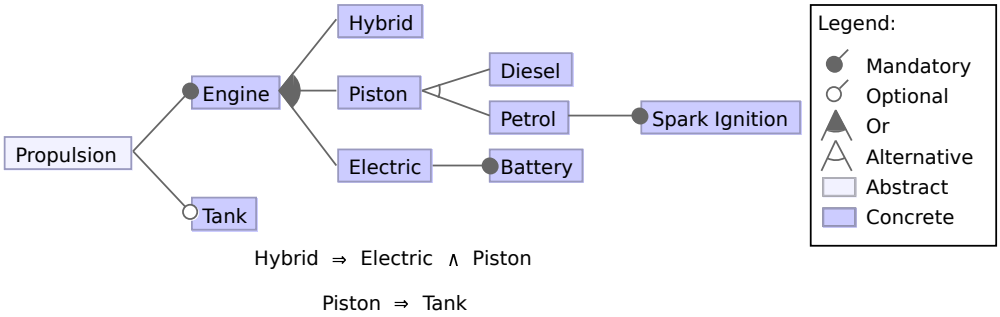


Fig. 2.1. Feature diagram for a feature model with eight valid configurations; two cross-tree constraints are specified as propositional formulas over features

2.2 Software Evolution

The first notion of the software development process is usually developer-centered and merely focuses on software being designed, implemented, tested, and eventually being released and deployed. Maintainability is a generally recognized software quality property to look after, and maintenance is, of course, essential to every successful software system (Liggesmeyer, 2009). Nonetheless, less attention is given to the ability to adapt a software system to changing requirements (evolvability) rather than maintaining it to keep functionality working (Parnas, 1994). Software evolution and evolvability, such as software itself are manifold. Software evolves in many ways ranging from maintenance (refactoring, bug-fixes, and patches) to adapting to changed requirements (adding, removing, reorganizing functionality, and variability). Modern software systems not only often ship with a variety of configuration options to select from, they also employ routines to be build and sometimes even make use of or are part of platforms, such as apps or plugins. That is, software evolution affects all aforementioned aspects, maintainability as well as evolvability can degrade as software evolves.

2.2.1 Software Erosion

The negative symptoms of software evolution which are referred to as “architectural erosion” (Breivold et al., 2012) have been addressed by many researchers. Most of

existing research so far though focuses on evolution with regard to software architecture (Breivold et al., 2012). The main driving factors leading to symptoms of decay identified by Perry and Wolf (1991) are architectural erosion and architectural drift. While architectural drift subsumes developers' insensitivity when not following a systems architecture or respective guidelines while making changes, architectural erosion subsumes ignoring and violating the existing software architecture. Parnas (1994) argues that as software evolves, software is maintained and evolved by developers who are not necessarily familiar with the initial architectural design. Therefore, knowledge about the architecture can become unavailable as software evolves. Although the unfavorable effects of software evolution do not break a system necessarily and imminently, the software becomes "brittle" (Perry and Wolf, 1991) as maintainability as well as evolvability degrade. Concrete symptoms of software erosion on the implementation level have been documented.
Zhang et al. (2013) have studied erosion symptoms for a large-scale industrial software product line with compile-time variability using preprocessor directives. The authors identify variability-related directives and clusters of those that tend to become more complex as the software evolves. The negative effects, or symptoms of software erosion are described as, but not limited to *code replication* and inter-dependencies between code elements, such as *scattering* and *tangling*. Code scattering describes the phenomenon of code belonging to a certain feature being scattered across multiple units of implementation, such as modules, whereas code tangling means that code from different and potentially unrelated features in entangled within a single module. A figurative and recapitulative term to refer to the trade-off between postponed maintenance or QA tasks and the corresponding cost is *technical debt* (Guo et al., 2011); the metaphor summarizes the growing risk of quality degradation for eroding software.
Passos et al. (2015) have studied the extent of usage of scattering for device-drivers in the Linux kernel. Despite scattering being quite prevalent, their findings suggest that the kernel architecture is robust enough to have evolved successfully. Nonetheless, platform drivers in the Linux kernel seem more likely to be scattered than non-platform drivers. They conclude that this is a trade-off between maintainability and performance: a more generalized and abstract implementation for platform-drivers in this case could possibly avoid scattering, yet refactorings in this manner did not seem to be necessary or worth the effort yet.

2.2.2 Variability Evolution

Apart from architecture evolution, the variability offered by software systems evolves as well. For configurable software systems, evolution steps will not only affect artifacts in the solution space, yet also be visible in changes in the respective variability models in the problem space. Although the variability aspect of software evolution has not gathered as much attention as architecture in the past, more and more research has emerged recently to address and understand variability evolution.
Peng et al. (2011) proposed a classification of variability evolution patterns that conceives evolution as adaption to changing (non-)functional requirements and contexts. For a context in that sense, two categories exist. A driving context determines, whether a variability model and respective variants can meet functional requirements in the first place. A supporting context by definition determines how non-functional properties are

strengthened or weakened. Any changed requirement is likely to change the contexts for a software systems variability model and, therefore, will make adaptations of the variability model necessary. Within their classification method, Peng et al. (2011) identify major causes for variability evolution, comprising (a) new driving contexts emerging, (b) weakened supporting contexts (for instance, due to new non-functional requirements), and (c) unfavorable trade-offs for non-functional properties.
To understand single evolutionary steps, several catalogs of variability evolution patterns have been proposed. Peng et al. (2011) present three patterns, where either a new feature is added, a mandatory feature becomes optional, or a mandatory/optional feature is split into alternative features. Seidl et al. (2012) suggest a catalog of patterns for co-evolution of variability models and feature mappings that additionally introduces code clones, splitting a feature into more fine-grained sub-features, and feature removal as evolution patterns. In addition, Passos et al. (2012) have studied variability evolution in the Linux kernel and present a catalog of patterns where features are removed from the variability model, but remain a part of the implementation. Their catalog, among others, includes feature merges, either implicit (optional feature merged with its parent) or explicit.
The classification proposed by Peng et al. (2011) is a general and formalized approach that, as well as Seidl et al. (2012) and Passos et al. (2012), describes elementary evolution patterns which can be composed to more complex patterns. Nonetheless, no comprehensive catalog of variability evolution patterns so far has been proposed.

2.3 Assessing Performance

While the last three sections covered software evolution and variability modeling, we now step forward to the topic of software performance. This section will outline the term performance with respect to software systems as well as to possible measurements. We provide a brief look at the general performance testing setup and the required prerequisites, including suitable benchmarks.

2.3.1 What is Performance?

The performance of software systems is, like software quality, primarily a matter of perspective. While an end user might consider practical aspects to be more important, from a developer's perspective, performance relates to and is best described by non-functional properties (Liggesmeyer, 2009; Molyneaux, 2014). While functional properties subsume what exactly a software system does, non-functional properties describe how a software system is at providing the functionality offered (Liggesmeyer, 2009). The notion of good and bad in this sense corresponds to non-functional requirements (NFR), that is, software with "good" performance behavior does not violate its NFRs. The categories of NFRs that shape performance behavior are manifold. According to Molyneaux (2014), the categories or *key performance indicators* (KPIs) include

- availability
- response time,
- throughput,

- resource utilization, and in a broader scope also
- capacity.

Time-related KPIs are availability and response time, whereby availability describes the time or time ratio that the software is available to the end user, and response time subsumes the time it takes to finish a request or operation. Throughput as a category subsumes the program behavior with respect to program load, such as hits per second for a Web application or amount of data processed per second. Resource utilization describes the extent to which a software system uses the physical resources (CPU time, memory, and disk, or cache space) of the host machine. Finally, from a Web-centered perspective, capacity describes measurements with respect to servers and networks, such as network utilization (traffic, transmission rate) and server utilization, such as resource limitations per application on a host server (Molyneaux, 2014).
Consequently, the assessment of performance requires a context or testing target that corresponds to the assessed system under test (SUT). For instance, for a simple command-line compression tool, suitable KPIs are response time and throughput, whereas performance for an online shop Web application is better outlined by availability and capacity.

2.3.2 Performance Testing

The first step in performance testing, prior to defining relevant KPIs and metrics, is to specify a system operation or *use case* (Woodside et al., 2007) to assess performance for. A typical use case includes a well defined task or workload to process, expected behavior, outcome, and performance requirements as previously discussed. For the SUT, however, we require a version that does compile or, in case it is interpreted, is syntactically correct (Molyneaux, 2014). With regard to performance assessments as part of the development process, a code freeze should be obtained since measurement results are likely to become meaningless for later versions. In addition to that, the machine or setup used for performance measurement should ideally be as close to the production environment as possible, but at least be documented to compare different runs (Molyneaux, 2014).
Finally, one or more benchmarks need to be selected to simulate the program load for the respective use case. A benchmark, all in all, needs to be representative, i.e., should relate to the use case or requirement one wants to validate. While benchmarks for file compression usually include multiple different types of media data (text, sound, pictures) such as the Canterbury corpus[1], Web applications can be exposed to handling with a number of simulated users at the same time (Molyneaux, 2014). Benchmarks have often been standardized within research and engineering communities to provide comparable performance measurements. A popular example is the Software Performance Evaluation Corporation (SPEC), a consortium providing a variety of benchmarks such as the CPU2000[2] processor benchmark consisting of programs with both floating point and integer operations. Moreover, benchmarks, in a sense of repeatable program load, can be obtained from load generation software such as ApacheBench[3]

[1]The Canterbury corpus can be found here, `http://corpus.canterbury.ac.nz/`.
[2]The benchmark description can be found at `https://www.spec.org/cpu2000/`.
[3]Manual page of the Apache tool `ab`, `http://httpd.apache.org/docs/2.4/en/programs/ab.html`.

for the Apache Web server or simply by measuring performance for test cases (Heger et al., 2013; Nguyen et al., 2014b).
Performance testing heavily relies on tool support, especially for repeating test cases and recording measurements. Since the tool solutions for performance testing vary from domain, scale, and purpose, we will outline only the general tool architecture. Molyneaux (2014) describes four primary components for a performance testing setup: a scripting module, a test management module, a load injector, and an analysis module. A scripting module handles the generation or repetition of use cases which, for instance, can be recorded prior to the test for Web applications. A test management module creates and executes a test session, whose program load is generated by one or more load injectors. A load injector can provide a benchmark by generating items to process, or can simulate a number of clients for a server-side application. An analysis module, finally, collects and summarizes data related to the performance testing target. We present more on summarizing and comparing recorded results in the sections 5.3 and 5.4.

2.4 Performance Prediction Models

In the previous section, we referred to performance, or in detail, the KPIs, as possible testing targets, which we validate against non-functional requirements. However, in a broader sense, software performance has become an aspect of software engineering referred to as *software performance engineering* (SPE) (Woodside et al., 2007). A lot of effort has been spent to study and describe performance behavior as well as to improve performance quality. Besides the analysis of concerns or requirements with respect to performance, SPE comprises performance testing as well as performance prediction (Woodside et al., 2007). Performance prediction aims at modeling and estimating performance behavior for different use cases or configurations. The first approaches to performance prediction models describe the underlying software system component or operation from which performance estimations are then deduced. These so-called *model-based prediction models* enable a performance estimation early in the development process since no actual performance measurement is required (Woodside et al., 2007). In contrast to model-based approaches, measurement-based approaches have emerged. These *measurement-based prediction models* are based on a sample of performance measures which are used to learn a software system's performance behavior (Woodside et al., 2007). More recently, measurement-based prediction models that emphasize variability have been proposed, of which we will discuss three approaches in the remainder of this section.

Learning Performance In essence, learning and predicting performance behavior for a configurable system means nothing different than finding an approximation $\hat{f}$ for a function $f : C \to \mathbb{R}$ where C is the set of configurations and $f(c) \in \mathbb{R}$ with $c \in C$ is the corresponding performance measurement. The accuracy of the approximation $\hat{f}$ describes how the estimated performance $\hat{f}(c)$ deviates from the actually measured performance $f(c)$. Different approaches to construct such an approximation, referred to as a *performance prediction model*, emerged recently. All approaches utilize two samples of configurations and corresponding performance measurements to build and

validate the model. A training sample is used to learn the approximation from, whereas a testing sample is used to assess the prediction error rate of the previously learned approximation.
A straightforward methodology to learn performance behavior was proposed by (Guo et al., 2013). The authors utilize classification-and-regression-trees (CART), akin to decision trees, to derive a top-down classification hierarchy for a given sample. The approach supports progressive (and random) sampling, i.e., the performance model is constructed several times while the size of the training sample is successively increased. The CART is constructed by recursively partitioning the sample into smaller segments which best describe a class. It is worth mentioning that the estimation using CART is not limited to binary configuration options, but supports numeric features as well. Moreover, the approach by (Guo et al., 2013) does not produce any further computation overhead besides the measurement effort and the construction of a CART.
A different approach to modeling performance behavior was proposed by Zhang et al. (2015). The authors propose an approach to construct performance prediction models based on a Fourier description of the performance-describing function f, which maps each configuration to a corresponding performance measurement. The principal idea is to approximate a Fourier series approximating the function f, and learning all coefficients of the Fourier series terms. The number of terms is exponential in the number of configuration options. The main characteristic of this approach is that, prior to learning a performance prediction model, a desired level of prediction accuracy can be specified. That is, the approach automatically chooses the sample size required to learn the prediction model accordingly.
A third approach to model and predict performance behavior, *SPLConqueror*, was proposed by Siegmund et al. (2012). The authors describe performance behavior for a configuration as the accumulated influence of features, from which the configuration is composed, and respective feature interactions. To estimate the influence of single features and feature interactions, the authors propose a number of sampling strategies. Single features are assessed by comparing the performance of two different configurations per feature: For a feature f, two valid configurations are compared, whereby both configurations have the minimal number of features selected that are not excluded by the selection of f. While for one configuration f is selected, it is deselected for the other one. The difference between the performance measures for both configurations is the estimate $\Delta_{min(f)}$ for the performance influence of feature f. Feature interactions that are performance-relevant are detected automatically in a similar manner. The main idea is, that a feature f is more likely to interact with other features, the more features are selected along with f. Therefore, if f interacts, the influence of feature f, $\Delta_{min(f)}$ differs from the influence of f, when estimated using configurations where the maximal number of features selectable together with f are used. Based on the set of interacting features, three heuristics are employed to detect feature interactions. Simple feature interactions are detected using pair-wise sampling (Siegmund et al., 2012; Apel et al., 2013), whereas higher-order feature interactions are often composed from simpler ones, or centered around a small subset of hot-spot features (Siegmund et al., 2012). Finally, for an arbitrary configuration, the performance can be predicted by the sum of influence estimations per feature and feature interaction.
The idea of predicting performance by estimating the influence of feature and feature interactions was further developed by Siegmund et al. (2015) by proposing performance-

influence models for both binary and numeric configuration options. A performance influence, similar to *SPLConqueror*, predicts performance by computing the sum of previously learned influence estimates. The novelty of this approach lies in the way the terms describing the influence of feature and feature interactions are learned and not derived from a sampling strategy. Instead of a single constant performance influence estimate, each term can contain of a number of functions, such as linear, quadratic, or logarithm functions, or compositions thereof. This does not only allow the developer to incorporate domain knowledge by preselecting functions. Moreover, but also by introducing combinations of non-linear functions and learning the coefficients using linear regressions, it enables learning a non-linear function. The algorithm commences with the selection of terms and successively extends and reduces the term selection to increase the prediction accuracy. The outcome, similar to Siegmund et al. (2012), is a linear function whose terms represent the estimated influence of features and feature interactions on performance.
All aforementioned approaches rely on performance measurements and, to some extent, also on a sampling strategy. While the approaches by Siegmund et al. (2012, 2015) essentially describe sampling strategies, the approach proposed by Zhang et al. (2015) is able to work with random samples, and the approach of Guo et al. (2013) allows a smaller sample to be extended progressively (Guo et al., 2013). As performance measurements entails an expensive and time-consuming process, Sarkar et al. (2015) have investigated different sampling strategies in this domain. The authors compare progressive sampling, where sample sizes are increased successively until the learned prediction model performs accurately enough, and projective sampling, where the optimal sample size is estimated based on the expected learning curve, with respect to cost and prediction accuracy. They advocate the use of projective sampling over progressive sampling. In addition, the authors propose and evaluate an own sampling heuristic to select an initial sample for progressive sampling. This sampling heuristic is based on feature frequencies and outperforms t-wise sampling, such as pair-wise sampling (Sarkar et al., 2015).

2.5 Related Work

While to the best of our knowledge, there exists no previous work on exhaustively presenting a methodology to understand and comprehend performance evolution of configurable software system, there is a body of work that has either touched on similar topics, or pursues a similar methodology, yet with different basic premises. In the following, we present and summarize related work to emphasize the limited scope of our work, and to give an outline of possible future research directions. Further note that we describe here only additional approaches to the ones we have already discussed in the previous sections.

Visualization of Software Evolution. Throughout our methodology, we have frequently used diverse visualizations of data obtained from repository mining, for instance, for commit activity, and as a basis for revision sampling strategies. In addition, we have taken into account similar data in the interpretation of our performance evolution history data in section. Therefore, as software evolves it is inevitable to consider

those data to obtain more complete and integrated insights. Aggregation and visualization of aforementioned data records can help to sketch and understand possible coherences.
German and Hindle (2006) have presented a comprehensive tool, *softChange*, to visualize multiple aspects of a software system's evolution history, including commit and file activity graphs, authorship overviews, and visualizations of file coupling. While their tool is not designed with regard to a specific research direction, the authors stress the importance of means to aggregate and analyze software trails in order to explore and understand software evolution. Joint visualizations of software trails as well as performance evolution history data is a promising augmentation in the comprehension of multi-layered software evolution.
Wu et al. (2004) have presented presented an integrated approach to visualize multiple different measurements over time. Their proposed *evolution spectrum graphs* are inspired by spectrum visualizations for audio signals. A spectrum in their context can, for instance, be a list of files, for which different measurements are illustrated over time. While this approach allows for a global overview over an arbitrary time span, it also can sketch fine-grained changes and trends over time. Although, according to the authors, the visualizations need to be tailored to the target system with respect to coloring and spectrum selection, evolution spectrum graphs can be a useful means to aggregate and visualize various single-valued measurements for a spectrum of variants for configurable software systems.
To conclude with a broader overview on visualization tools for further reading, Storey et al. (2005) have surveyed twelve tools (including the previously mentioned two) that intend to visualize any kind of human activities in software development with a string focus on software evolution. The authors evaluate the different tools with respect to different aspects, including intent of the tools, the information sources utilized by the tools, the form of presentation offered by the tools, and the effectiveness in terms of feasibility and validity. We see these lines of work orthogonal to our methodology such that we can incorporate different visualization techniques when it comes to evaluating and understanding the performance evolution of configurable software systems.

Power Consumption Evolution Hindle (2015) have proposed a methodology, *green mining*, to measure and model the evolution of power consumption across different versions of a software system. Using their methodology, the authors have documented the power consumption evolution history of two different software systems, and identified feasible metrics to summarize power consumption.
Compared to our proposed methodology, green mining considers only one-dimensional evolution of power consumption as different software variants are not taken into account, yet green mining examines the relationship between power consumption and further quality attributes. That is, we believe that interpreting performance evolution results in an extended context of software metrics with respect to software architecture, power consumption, and along established software analyses should direct further research activities towards a better understanding of software evolution.

3 Methodology: Variability Assessment

To assess performance evolution for configurable software systems, it is required to assess and understand variability of such systems with respect to various aspects: *First*, to actually assess single variants, knowledge about the variability model is required to configure the software systems accordingly. *Second*, obtaining knowledge about feature usage and implementation can provide meaningful insights. For instance, knowing that most variable code is dependent on small numbers of features might be useful information when selecting a sampling strategy. Similarly, knowing which feature combinations are frequently involved in conditioning program functionality and behavior can help understand configuration-related defects. *Last*, since the number of configurations for most configurable software systems is infeasible, variability assessment faces performance- and scalability-related problems. That is, variability assessment is usually constrained by limited resources.

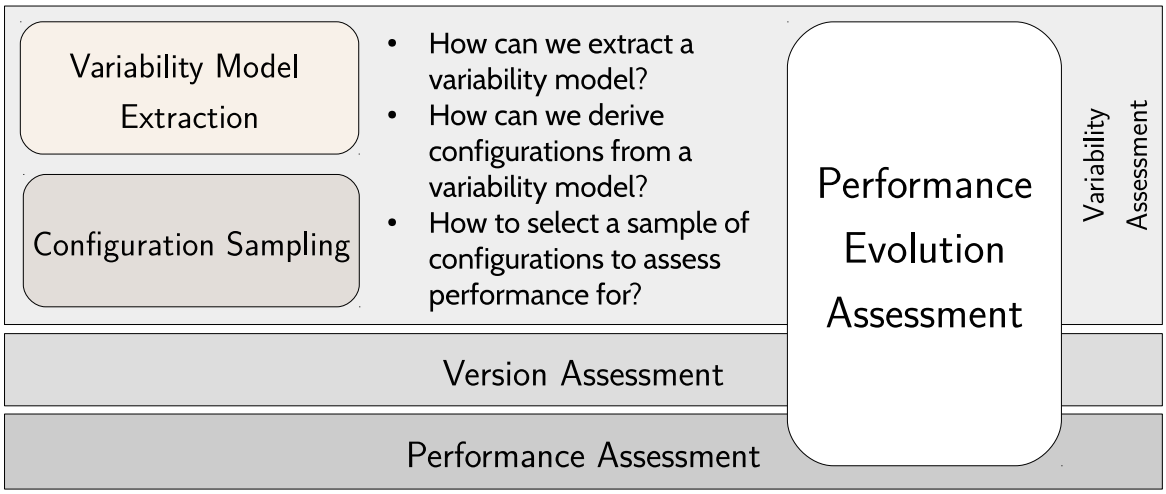


Fig. 3.1. Methodological road-map: questions to address during variability assessment.

This chapter describes the first aspect in our methodology, the assessment of variability. As illustrated in Figure 3.1, this first tier of our methodology embraces two main tasks, first, the synthesis (or extraction) of a variability model for a configurable software system, and second, strategies to select meaningful sample sets of configurations to assess performance for. In section 3.1 we review the synthesis of variability models as well as corresponding analyses of systems' variability; in section 3.2 we describe means to generate configurations from variability models, and finally, section 3.3 discusses different strategies for selecting sample sets of configurations.

3.1 Untangling Variability

Before presenting strategies to synthesize variability models in the next section, we first recap a number of techniques to untangle configurable software composed from variable and invariant code. In the context of our methodology, these techniques are useful means to make variable code segments visible, and to understand how variability influences the overall resulting variants of a software systems. The idea of designing and developing configurable software systems is driven by the separation of different concerns, expressed as features of a software system. A configurable software system is either assembled at compile-time with respect to its configuration, or tailored to a configuration at load-time (Apel et al., 2013). Both ways of implementing the variability exist in practice. Research has also proposed a variety of variant-generating implementation techniques for compile-time variability, ranging from simple preprocessor directives to features as separate modules (Kästner et al., 2009). A complete survey of means to implement configurable software systems would likely exceed the scope of this section, yet we intend to discuss in this section which information regarding variability is required or useful to our methodology, and how it can be obtained.

3.1.1 Family-based Analyses

As the number of variants for configurable software systems is usually infeasible, naive analyses of configurable systems are not trivial. Any static analysis can only assess one variant at a time, as well as dynamic analyses, which can follow only one execution path. In contrast to that, recently, extended analysis techniques, which are aware of variability of the systems studied, have emerged (Thüm et al., 2014). In particular, these *family-based* analyses avoid redundant computation, such as visiting a code section multiple times, and exploit artifacts shared by multiple variants (Thüm et al., 2014). Besides more efficient analysis, family-based methods incorporate knowledge about valid feature combinations (Thüm et al., 2014) and, therefore, connect analysis results with a context, such as feature combinations, for which the findings hold. Family-based methods have been widely used across various domains and can provide useful information when assessing configurable software systems.

Variability-aware Parsing. Kästner et al. (2011) have proposed the framework *TypeChef* to enable the construction of variability-aware parsers. A variability-aware parser, like ordinary parsers, systematically explores a program to return an abstract representation of the parsed program. This parse tree, or abstract syntax tree, is the basis for compilers to translate a program, or for further static analyses, such as type checking. For a code base with variability expressed by preprocessor annotations, which are evaluated prior to compilation, a variability-aware parser, however, is able to derive a parse tree considering all variants in a single run. A parse tree usually consists of nodes representing syntactical features of the parsed program. The parse tree returned by a variability-aware parser, moreover, comprehends variable segments of a program and will include them with respect to their presence conditions. For instance, a class may contain a function *sort()*, for which two different implementations exist. While there might be numerous variants, the parse tree of the class will contain a node with two children, one for each implementation; higher numbers of alternative implementations

are expressed by nesting further nodes. In that way, variable and invariant program segments can be separated.
While the approach of Kästner et al. (2011) handles undisciplined usage of preprocessor directives, such as splitting function parameter lists, variable types, or expressions, Medeiros et al. (2017) have proposed an approach to avoid and conservatively refactor those cases. The authors propose a catalog of refactoring templates, which describe transformations from undisciplined usage of preprocessor annotations to disciplined ones. With respect to variability-aware parsing, disciplined usage is conceived as using preprocessor annotations only to segment statements, but not to segment a single syntactical unit, such as expressions (Medeiros et al., 2017).
In the context of our methodology, the parse tree resulting from variability-aware parsing can be used as a basis for further analyses. Since the parse tree provides a machine-readable decomposition of variable and invariant code segments along with presence conditions, for instance, it has been used to derive constraints among features (Nadi et al., 2014, 2015) as we will see in the next subsection.

Staged Variability. Besides variability-aware parsing, Nguyen et al. (2014a) have applied symbolic execution (King, 1976; Darringer and King, 1978) to unwind variability for PHP Web applications. Web applications are staged, i.e., even though they can be configured at load-time, the application is as well variable with respect to input received at run-time. For instance, consider *WordPress*, a popular content management system (CMS) implemented in PHP, which can be extended with a number of plug-ins. However, the content of a website presented to the user also depends on information retrieved from a database, and user input. Consequently, a dynamic PHP Web application is staged in a sense that it generates configurable HTML templates which are rendered at run-time. The authors utilize a symbolic execution engine to explore all possible execution paths. Each user input or database query is considered a symbolic value which is propagated through each script. By keeping track of the (partially symbolic) HTML output and organizing it in a DOM-like structure, their approach approximates the HTML output, which subsequently can be tested, for instance for validity (Nguyen et al., 2011). Similarly, Lillack et al. (2014) have applied taint-analysis to configurable software systems to track the influence of configuration options read at load-time. Their static analysis approach taints every value resulting from reading a configuration parameter as well as every value resulting from a computation that involves previously tainted values. That way, lines of code that are possibly depending on configuration options are detected.
In the context of our methodology, addressing staged variability with techniques such as symbolic execution is required to assess performance for Web applications. Both the code executed at server-side as well as the dynamically generated HTML are interdependent parts of the Web application. Any suitable and elaborate performance measurement setup for Web applications must incorporate both stages.

Build System Variability. Apart from configuring software systems using preprocessor annotations, the assembling of a configurable software system can as well be orchestrated by its underlying build system. While preprocessor annotations virtually separate code fragments of different features, for instance, build systems can physically exclude files from compilation. This implementation of variability enabled by

build systems, in particular of Makefiles, has been subject to a couple of analysis approaches. Tamrawi et al. (2012) have proposed *Symake*, a symbolic execution engine to evaluate Makefiles. On top of *Symake*, Zhou et al. (2015) utilize symbolic execution to analyze Makefiles and derive file presence conditions, stating under which feature selection a file is included or excluded from compilation. The work of Al-Kofahi et al. (2016) addresses a more advanced build system, *GNU Automake*. *Automake* describes a staged build process, where a Makefile can be specified on a higher level, and is subsequently compiled to an actual Makefile. The authors' aim to provide a variability-aware representation of all possible Makefiles to enable further analyses of the build process. In the context of out methodology, the symbolic evaluation of a software system's build process follows an intention similar to the one of variability-aware parsing. Not only could further analysis tools be build upon a symbolic evaluation engine, but also does symbolic evaluation in this context untangle the variability accommodated in the build process. The file presence conditions extracted by Zhou et al. (2015) and Al-Kofahi et al. (2016) are a strong basis for further constraint extraction, and therefore variability model assessment.

In the context of our methodology, the mentioned family-based analyses serve two fundamental purposes. *First*, family-based analyses can help to obtain an overview of what, or how much code in a configurable software system is variable as well as to which extent. For instance, if most extracted presence conditions contain only very few features, higher-order feature interactions are unlikely. *Second*, family-based analyses provide a feasible means to untangle variability and, as we will see in the next section, are the basis for some elaborate techniques to extract variability models (Nadi et al., 2014, 2015). In conclusion, these analyses are not essential to extract variability models, yet are useful tools to use in addition to the strategies mentioned in the next section.

3.1.2 Variability Model Synthesis

A variability model as an abstraction of functionality of a software system is required, or at least of great interest, in many contexts. Not every configurable system provides an explicit representation of its variability model. The reasons for inexplicit or absent configuration specification are manifold. They can range from poor or inconsistent documentation (Rabkin and Katz, 2011) to overly complex configurability (Xu et al., 2015), or configuration constraints originated in different layers of a software system, such as build constraints or compiler constraints (Nadi et al., 2014, 2015). The following paragraphs review different *strategies to synthesize variability models* from different types of artifacts. A classification of strategies, along with corresponding literature is illustrated in Figure 3.2.
The first category comprises extraction based on Natural Language Processing (NLP) utilize similarities between textual representations of different variants to derive common features (Alves et al., 2008; Bakar et al., 2015). The approaches in the second category are heuristics based on static analyses, whereby configuration option names (Rabkin and Katz, 2011) are extracted, or constraints are derived from assessing the software's build process (Nadi et al., 2014, 2015). The last category comprises techniques which conceive variability model as an optimization problem. For a set of given and valid configurations, genetic algorithms are used to approximate an optimal fea-

ture model representing constraints among features (Lopez-Herrejon et al., 2012, 2015; Linsbauer et al., 2014).

NLP-based Extraction. As feature diagrams group and organize features (representing functionality), synthesizing a variability model has shown to be applicable to extract features and constraints from natural language artifacts. For instance, by comparing product specifications for an existing market domain, variability models can provide a detailed feature summary (Alves et al., 2008).
The basic idea is to identify commonalities and differences in natural language documents, such as product descriptions, requirements, or documentations, by using natural language processing (NLP) techniques (Bakar et al., 2015). A widely employed technique is to conceive a text as a vector in a vector space model, where each word or token represents a dimension. From the tokenized text, irrelevant stop words are removed, and all remaining words are reduced to their original word stems. The importance of all remaining tokens is (usually) weighed by its tf-idf value, an established technique in information retrieval. That is, each text (corresponding to a variant or configuration) is represented as a vector of tf-idf values in the aforementioned vector space model. Based on these representations, text instances are clustered to identify commonalities and differences, for instance in terms of shared words. Subsequently, the clustering information can be used to extract features or entire feature models (Alves et al., 2008; Bakar et al., 2015).

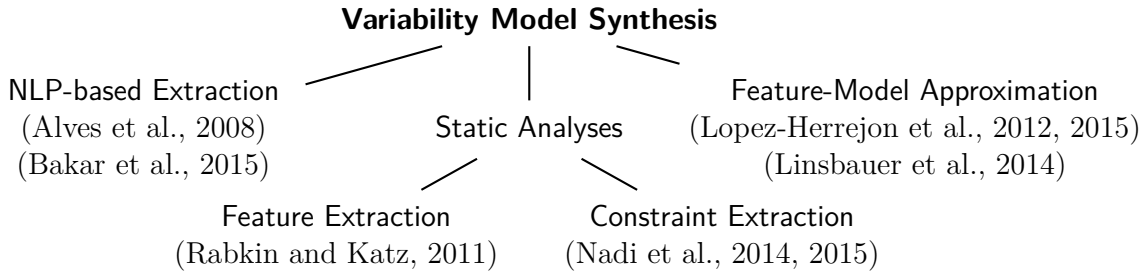


Fig. 3.2. Overview of our literature survey on variability model synthesis.

Feature Extraction. Rabkin and Katz (2011) proposed a static, yet heuristic approach to extract configuration options along with respective types and domains. Their approach exploits the usage of configuration APIs and works in two stages. It commences with extracting all code sections where configuration options are parsed. Next, configuration names can be recovered as they are either already specified at compile-time or can be reconstructed using string analysis yielding respective regular expressions. Moreover, the authors employ a number of heuristics to infer the type of parsed options as well as respective domains. *First*, the return type of the parsing method is likely to indicate the type of the configuration option read. *Second*, if a string value is read initially, the library method it is passed to can reveal valuable information about the actual type. For instance, a method `parseInteger` is likely to parse an integer value. *Third*, whenever a parsed configuration option is compared against a constant, expression, or value of an enum class, these might indicate valid values or at least corner cases of the configuration option domain. The extraction method by Rabkin

and Katz (2011) is precise, but limited, for instance, when an option leaves the scope of the source code and is passed to a database. Nonetheless, for the systems studied, the authors were able to recover configuration options that were not documented, only used for debugging or even not used at all.

Constraint Extraction. A more comprehensive investigation of configuration constraints and their origin is provided by Nadi et al. (2014, 2015). They use variability-aware parsing to infer constraints by evaluating Makefiles and analyzing preprocessor directives. Inferred constraints result from violations of two assumed rules, where (a) every valid configuration must not contain build-time errors and (b) every valid configuration should result in a lexically different program. While the first rule aims at inferring constraints that prevent build-time errors, the second one is intended to detect features without any effect, at least as part of some configurations. Their analysis has shown a high accuracy in recovering constraints with 93 % for constraints inferred by the first rule and 77 % for second one respectively. However, their approach recovered only 28 % of all constraints present in the software system. Further qualitative investigation, including developer interviews, lead to the conclusion that most of existing constraints stem from domain knowledge (Nadi et al., 2015).

Feature-Model Approximation. A different strategy to recover variability models, instead of analyzing the software artifacts, is to approximate a model. Given a selection of valid feature selections, a variability model best describing the configurations can be approximated, or learned. Lopez-Herrejon et al. (2015) have surveyed different search-based strategies to synthesize feature models of which we present two categories. Evolutionary algorithms have been applied to reverse engineer feature models from configuration samples (Lopez-Herrejon et al., 2012; Linsbauer et al., 2014). A population of feature models is generated and each instance is evaluated by a fitness function, measuring how well it fits the given sample set of configurations. Subsequently, a new generation is obtained by applying crossover and mutation operators to the previous generation, whereby only the fittest instances remain. This process of evolution is repeated multiple times until a desired threshold fitness is reached for a feature model instance. Lopez-Herrejon et al. (2012) identify a trade-off between the accuracy of recovered feature models and the number of generations employed by evolutionary algorithms. Besides promising results, the authors stress the importance of effective and scalable fitness functions as well as meaningful samples to learn the feature model from. Contrary to evolutionary algorithms, Haslinger et al. (2011, 2013) have proposed an ad-hoc algorithm to reverse engineer feature models. The algorithm recovers the feature model layer by layer via extracting all child features for a given parent feature recursively. The algorithm does not consider cross-tree constraints. Besides promising results for basic feature models, the authors advocate the incorporation of human domain-knowledge in the synthesis of feature models.

Feature-Hierarchy Recovery. Besides recovering features and their respective constraints, to reverse engineer a feature model, one further step is required when the outcome should be human readable in a feature diagram. The recovered knowledge can be organized in a tree-like hierarchy with feature groups specified and cross-tree constraints explicitly stated to derive a valid feature diagram (Kang et al., 1990). While

several approaches for recovering the feature-model hierarchy have been proposed, we are primarily interested in finding a hierarchy for knowledge obtained from source code. In the remainder of this subsection, we will focus on organizing features and constraints extracted from source code.
Given an extracted set of features along with corresponding descriptions and recovered constraints among the features, She et al. (2011) propose a semi-automated and interactive approach to synthesize a feature hierarchy. Their approach comprises three steps. *First*, an overall feature hierarchy based on feature implications is specified. *Second*, potential feature groups are detected and manually selected. *Finally*, the feature diagram is extended with remaining CTCs. The approach by She et al. (2011) provides a practical algorithm to synthesize a feature diagram, yet has some limitations we need to consider. *First*, the approach is not able to detect or-groups as defined in Sec. 2.1. *Second*, the approach does introduce a root feature. *Finally*, the approach does not distinguish between mandatory and optional features. Implicitly, all features that do not have a parent feature are optional and all features that have a parent feature are by default mandatory. She et al. (2011) evaluated the algorithm with both complete and incomplete variability knowledge (feature names, descriptions and constraints). While the algorithm has shown to be practical, detecting features whose parent was the root-feature was difficult since, due to the transitive property of implication, it is implied by each feature of the feature model.

Although the approaches mentioned in the paragraphs above, excluding the feature-hierarchy recovery, in principal address the same problem, they are isolated solutions to the problem of variability assessment, and their applicability is depending on a number of cross-cutting boundary conditions. *First*, for the overall problem of variability model synthesis we can identify two different contexts for which different techniques apply. The NLP-based techniques summarized by Alves et al. (2008) and Bakar et al. (2015) address the extraction of features in the context of requirements engineering, for instance by comparing different software products in the same market domain. However, the remaining techniques intend to extract features for a given single software system. *Second*, for variability assessment there exist two different principal analysis approaches. While the techniques proposed by Nadi et al. (2014, 2015) and Rabkin and Katz (2011) approach a configurable software system as a monolithic system, both the family of NLP-based techniques and the approximating techniques Lopez-Herrejon et al. (2012, 2015); Linsbauer et al. (2014) explicitly require a set of variants or configurations, respectively. *Third*, the techniques differ in the type of variability they are able to extract variability models for. While the approach of Nadi et al. (2014, 2015) only works for build-time variability, the work of Rabkin and Katz (2011) will only work for configurations read at load-time. Moreover, the remaining techniques do not consider the different types of variability at all.
In conclusion, it becomes clear that there is no single textbook solution to the problem of variability model assessment as this problem may arise in different contexts be approached from different perspectives, or be emergent for different types of variability.

Criterion	Scenario A	Scenario B	Scenario C
Configurability documented?	√	(√)	(√)
Configurations provided?	×	√	(√)
Variability model provided?	×	×	√

√ = Criterion satisfied, (√) = Criterion satisfied optionally, × = Criterion not satisfied

Tab. 3.1. Distinction of three scenarios for variability model synthesis.

3.1.3 Methodological Strategies

The last two subsections reviewed a number of family-based analyses for configurable software systems as well as approaches proposed to partially extract variability models from a system's code base. The latter approaches presented, however, are rather isolated solutions due to non-generic assumptions, such as the use of configuration APIs (Rabkin and Katz, 2011), or build-time variability (Nadi et al., 2015). At best, these approaches complement each other, or are an appropriate choice under certain circumstances, still requiring further manual assessment. The following integrates the previously discussed work and proposes methodological strategies to synthesize variability models for different scenarios, or use cases.

For the recovery of variability models, we propose three scenarios. Clearly, this is not an exhaustive list, but covers the majority of use cases based on our literature review. The scenarios should provide a practical context to the previously mentioned techniques. The three scenarios are outlined in Table 3.1; we derive our scenarios based on three criteria. *First*, we ask whether, and if so, to what extent configurability for a software system is documented. *Second*, we ask whether the software system provides sample configurations, such as configuration presets, or whether it ships as different variants, such as different Windows flavors. *Last*, we ask whether a variability model is explicitly contained in the software, and whether it is visible to practitioners, such as the *Kconfig* system for the Linux kernel. For each scenario, in Table 3.1, satisfaction of either criterion is marked. In addition, we mark criteria as *satisfied optionally*, if we assume them to be satisfied, but they are not necessarily relevant for the choice of strategy.

Apparently, the latter scenario C in Table 3.1 requires only little to no assessment of the application's variability since the variability model is already available.

For scenario B, despite documentation might be available, the previously presented variability model approximation approaches (Haslinger et al., 2011, 2013; Lopez-Herrejon et al., 2012; Linsbauer et al., 2014) can be applied, yet only binary configuration options are supported so far. Given the existence of sample configuration or configuration presets, these may provide additional information to answer the questions stated above. The remaining approaches mentioned in the previous subsections, unfortunately, describe only isolated solutions. Given suitable circumstances, they can nevertheless aid the extraction of variability models. The overall scheme in synthesizing a variability model is to answer the the questions above manually, and refer to automated tool support whenever possible.

For scenario A, when the main source of information regarding variability is the software system's documentation, we are left with no other option than manual assessment. To do so, we provide a questionnaire of five main questions to be answered in Table 3.2.

Question	What approach or technique to use?	Information Gain
How is variability implemented?	□ Manual review of the documentation with respect to what configuration mechanism is used, such as run-time parameters or preprocessor annotations. □ Family-based variability analyses help understand what parts of the software are variable and how they are enabled, or made accessible.	Knowledge of whether the software must be compiled for each configuration, or only for each version.
How can the software be configured? *Variability can be implemented in various ways.*	□ Manual review of the documentation with respect to how configurations are actually encoded to be machine-readable. Look out for standard or example configurations.	Knowledge about how a configuration can be translated to a machine-readable format required by the software, such as argument lists or preprocessor annotations.
Which configuration options exist, including types and domains? *Configuration options can be binary or numeric options.*	□ List all configuration options mentioned in the documentation. Unless not provided, consider standard or sample configurations or option names to infer a type, and valid values and corresponding domains.	Knowledge about what configuration options exist, whether they are binary or numeric options, and what valid values can be assigned to them.
Which dependencies or exist between and among configuration options?	□ Manual review of the configuration options with respect to implied, excluded, or alternative configuration options. Similarly, numeric constraints may exist.	Obtaining a (approximated) variability model, which is required to decide whether a configuration is valid or not.
Has the feature model been changed during evolution?	□ Review of release notes and changes in the documentation with respect to new features as well as constraints added, modified, or removed. □ Manual inspection of changes over time in code sections where configuration options are read.	Knowledge about whether synthesizing different versions of the variability model is required.

Tab. 3.2. Questionnaire for manual variability assessment.

The corners of the questionnaire cover, besides the manual synthesis of the variability model, the questions of how variability is implemented, configurations are encoded, and whether the variability model has changed during evolution. Although the automated approaches discusses earlier are only applicable for scenarios B or C, or under specific circumstances, family-based analyses can support manual assessment.
We have seen that the extent to which automated solutions are applicable to variability assessment depends on the degree to which variability is documented. If the variability model is available in a machine-readable format, little to no further assessment is required, while, if documentation was done manually, so variability assessment remains a task to be done manually. In conclusion, the questionnaire in Table 3.2 covers most aspects required to comprehend variability for a configurable software system and can be conceived as a guideline. Whenever applicable, additional synthesis or analysis techniques can be taken into account.

3.2 Configuration Generation

The next intermediate step in our methodology is the derivation of configurations from the variability model. Once obtained, we use the variability model to derive valid feature selections. For the assessment of quality attributes for configurable software systems, such as test case coverage or performance, we usually assess a sample set of variants. Hence, we require techniques to derive valid feature selections from the variability model to select meaningful sample sets from. As there exist various forms to express a variability model, configurations may be generated in various ways. Variability models can, for instance, be expressed as propositional formulas, context-free grammars (CFG) (Batory, 2005), or constraint satisfaction problems (CSP) (Benavides et al., 2005a,b). Accordingly, all configurations represent solutions to propositional formulas or CSPs, or valid words for CFGs respectively. That is, the generation of all configurations with respect to the variability model is equivalent to finding a solution or sentence for the aforementioned representations of a variability model. In the following, we review how variability models can be encoded as a CSP and describe in detail the configuration generation using CFGs.

3.2.1 Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) in the context of variability modeling describes a set of options ranging over finite domains as well as a set of constraints which restrict the value range of a variable (Benavides et al., 2005a). For a binary option b, the respective domain $dom(b)$ simply is $\{0,1\}$. For a numeric option, the respective domain $dom(n)$ is a finite set of legal values with a minimum and a maximum value, say $\{v_{min}, v_1, v_2, \ldots, v_{max}\}$. A solution $s : O \rightarrow dom(o_1) \times dom(o_2) \times \ldots \times dom(o_{|O|})$ to a CSP is an assignment of options $o_i \in O, i \in \mathbb{N}$ to values of their respective domain, such that all constraints are satisfied simultaneously (Benavides et al., 2005a).
A solution to a CSP is found by systematically checking for different selections of values whether all constraints are satisfied. There exists a large number of ready-to-use SAT and CSP solvers, yet we are not covering CSP solution here since it is beyond the scope of the thesis. For further reading, (Benavides et al., 2007) present a tool with extensive analysis support for various different presentations of variability models.

To encode a variability model as a CSP, Benavides et al. (2005a) describe the following transformation rules:

- For a parent feature f and a child feature f', a mandatory relationship is expressed as $f \Leftrightarrow f'$, and an optional relationship is expressed as $f' \implies f$.
- For a parent feature f and child features f_i, where $i = 1, 2, \ldots n$, an or-group is expressed as $f \Leftrightarrow f_1 \vee f_2 \vee \ldots \vee f_n$, and an alternative-group is expressed as

$$\bigwedge_{i=0}^{n} f_i \Leftrightarrow (f \wedge \bigwedge_{j \in [0,n] \setminus i} \neg f_j)$$

.

To also consider numeric options, the domain of a numeric option n can be conceived as an alternative-group since only one value from the domain can be selected at a time. This is often called disretization of the domain. Hence, each value of the domain $dom(n)$ can be conceived as a binary option. If value $v \in dom(n)$ is selected, this states $n = v$, otherwise $n \neq v$.

3.2.2 Grammar Expansion

Besides trying to find a solution for satisfiability problems, the expression of variability models as context-free grammars (CFGs) enables the derivation of configurations directly from a CFG by expanding it. A first description of transformation rules, yet only for feature diagrams with binary options, was was proposed by Batory (2005). A hierarchical feature diagram can be recovered from a set of constraints using the algorithm of She et al. (2011) as explained in section 3.1.2. In the following, we describe how a feature diagram with both binary and numeric features can be transformed to a context-free grammar, and how configurations can be derived subsequently.

Def. 3.2.1 (Context-free Grammar)**.** *A context-free grammar is a tuple $G = (N, T, S, P)$, consisting of a set of non-terminal symbols N, a set of terminal symbols T, a start word $S \in (N \cup T)^*$, and a set of productions $P \subseteq N \times (N \cup T)^*$. The set L_G describes the language of the grammar G and comprises all valid words $w \in L_G$ which can be derived from the start word $S \in L_G$ by applying productions a finite number of times to it.*

Following the Definition 3.2.1, to derive all configurations for a given feature diagram, the idea is to first translate it to a context-free grammar. In order to do so, especially with respect to handling numeric options, we introduce an extended definition for a CFG, a configuration generation grammar.

Def. 3.2.2 (Configuration Generation Grammar)**.** *A configuration generation grammar is a context-free grammar $G = (N, T, S, P)$ whose elements are constructed from a feature diagram as follows.*

- *All features represent non-terminal symbols N, which can be divided into two disjoint sets, binary non-terminal symbols $F_{\mathcal{B}}$ and numeric non-terminal symbols $F_{\mathcal{N}}$. That is, $N = F_{\mathcal{B}} \cup F_{\mathcal{N}}$ and $F_{\mathcal{B}} \cap F_{\mathcal{N}} = \varnothing$.*

- *Similarly, the set of terminal symbols T is consists of two different sets, the binary terminal symbols $T_{\mathcal{B}}$ and the numeric terminal symbols $T_{\mathcal{N}}$, so that $T = T_{\mathcal{B}} \cup T_{\mathcal{N}}$ and $T_{\mathcal{B}} \cap T_{\mathcal{N}} = \varnothing$ with*

$$T_{\mathcal{B}} = \bigcup_{b \in F_{\mathcal{B}}} \{b_0, b_1\} \tag{3.1}$$

$$T_{\mathcal{N}} = \bigcup_{n \in F_{\mathcal{N}}} \bigcup_{v_i \in dom(n)} \{n_{v_i}\}. \tag{3.2}$$

- *All productions P are constructed from the hierarchy specified in the given feature diagram, the binary, and the numeric features. In our definition of a configuration grammar, each word w is expressed as a subset of (non-)terminal symbols, i.e., $w \subseteq (N \cup T)$. A word is a terminal word, if and only if it does not contain any non-terminal symbol. Accordingly, the set of productions is $P \subseteq N \times (N \cup T)$, and a production $p = (u, v) \in P$ is applied to a word w by removing non-terminal symbol u from word w and merging words v and w. Hence, the new word w' is defined as $w' = (w \setminus u) \cup v$.*

  The productions $P = P_H \cup P_F$ are constructed from the following disjoint two sets of productions:

$$P_H = \{(p, \{c, c_1\}) | p, c \in N \wedge c \implies p\} \tag{3.3}$$

$$P_F = \bigcup_{f \in (F_{\mathcal{B}} \cup F_{\mathcal{N}})} \bigcup_{v \in dom(f)} \{(f, v)\} \tag{3.4}$$

- *Finally, the start word $S \subseteq (N \cup T)$ consists the non-terminal representing the top-level feature in the given feature diagram. The set of respective configurations is described by all words which can be generated by a finite number of applications of productions to the start word S.*

Based on Definition 3.2.2, we can specify an algorithm that computes the transitive closure of the grammar by repeatedly expanding each non-terminal for each (partial) word until a word containing non-terminal symbols is left. In addition to deriving all configurations from a grammar, the algorithm can also be used to derive partial configurations, such as binary-only configurations. To do so, the numeric features need to be removed from the set of non-terminal symbols. The only limitation of this algorithm is that, conceptually, it requires all numeric options to be mandatory features. This is due to the unspecified semantics of a numeric option being un- or deselected.

In the context of our methodology, configuration generation remains an intermediate step between the synthesis of a variability model and the selection of a meaningful sample set of configurations. The results obtained from applying both techniques, the encoding as a logical problem or the translation to a context-free grammar, to a variability model are equivalent. However, both techniques differ in terms of cost and tool support. While the former technique generally demands additional tool support, such as SAT or CSP solvers, their use is well established among research tools, such

as *FeatureIDE* (Al-Hajjaji et al., 2016) or *SPLConqueror* (Siegmund et al., 2012). Moreover, research has shown that SAT solvers generally scale for large configurable software systems. In turn, the latter technique using context-free grammars can easily be implemented, yet the exhaustive expansion of a context-free grammar is infeasible and does not scale since all valid configurations are derived. Nonetheless, for handling smaller variability models, context-free grammar might be a makeshift solution.

3.3 Configuration Sampling

When assessing emergent properties for configurable software systems, it is infeasible to consider every possible variant. As previously stated in section 2.1, interactions between features can emerge, and can be the root cause for configuration-related performance-interactions. Hence, effects on performance quality may be identified only under specific configurations. To not exhaustively assess all variants, a variety of strategies to select a sample set of configurations have been proposed. Every sampling strategy in the context of configurable software system is designed with respect to a *coverage criterion* (Apel et al., 2013). While some coverage criteria take into account the coverage of feature interactions, such as t-wise sampling (Williams and Probert, 1996), others consider code coverage, such as statement coverage sampling (Tartler et al., 2014). Although configurations in our case can contain both binary and numeric options, we distinguish sampling strategies for both categories. In the following, we review a selection of popular sampling strategies for binary options as well as sampling strategies for numeric options, especially in the context of performance assessment.

Binary Features. Most sampling strategies in the context of configurable software systems target binary features. Popular sampling strategies for sampling configurations of binary features include, but are not limited to the following (Apel et al., 2013; Medeiros et al., 2016) strategies listed in Table 3.3.
Medeiros et al. (2016) have compared different sampling strategies, among other things with respect to resulting sample size and fault detection rate. *Most-enabled-disabled* results in the smallest sample size, whereas *t-wise* sampling, especially for a greater t yields the largest samples. Regarding the detection of faults, *statement-coverage* performed poorly, whereas *t-wise* sampled samples, especially with a greater t unveiled most faults. Note that sampling with respect to statement coverage is not applicable in the context of performance assessment since for performance measurements, the software system is generally conceived as a black box.

Numeric Features. Similar to selecting meaningful sample sets of binary options, for numeric options, sampling strategies are designed with respect to covering possible interactions. Subsumed under the term *design of experiments* various sampling strategies, or experimental plans have been proposed, each assigning values (from an option's domain) to independent variables (numeric options, in this case) (Antony, 2014). As the choice of an experimental plan (for both binary and numeric options) determines the number of measurements, and therefore the cost of assessment, not all designs might scale and be suitable for our assessments. Siegmund et al. (2015) have reviewed and evaluated a number of established experimental plans with respect to applicability in

Strategy Name	Description
Feature Coverage	Configurations are selected, so that each feature is selected in at least one configuration (Apel et al., 2013). An extension proposed by Sarkar et al. (2015) for projective sampling takes into account feature frequencies. The frequency of a feature is the frequency of the feature being selected or deselected in a sample set. As a coverage criterion, a specified minimum feature frequency is to be reached.
Most-enabled-disabled	Configurations are selected, so that a maximal and minimal number of features is enabled (Medeiros et al., 2016).
One-enabled	Configurations are selected, so that each feature is enabled at a time (Siegmund et al., 2012). Similarly, with a *one-disabled* strategy, configurations are chosen, so that each feature is deselected at a time.
Random Sampling	Configurations are selected randomly, i.e., for a configuration, for each optional feature a random value out of 0 or 1 is assigned. Guo et al. (2013) have studied learning performance of a configurable software system from a random sample with acceptable accuracy.
Statement-coverage	Configurations are selected, so that each variable block of code (for compile-time variability) is at least included in one variant (Tartler et al., 2014). This either applies to variable code sections via preprocessor annotations, or to entire files and compilation units.
T-wise sampling	Configurations are selected, so that all t-tuples of all (binary) features are included in at least one configuration (Williams and Probert, 1996). That is, the upper bound for the sample size is $\binom{\lvert F\rvert}{t}$ for features F and $t \in \mathbb{N}$. Siegmund et al. (2012) extend this approach by composing higher-order feature tuples ($t > 2$) from already known pair-wise interactions. The rationale behind this heuristic is that, if pairs of features interact, also tuples including those pairs are likely to interact.

Tab. 3.3. Selection of sampling strategies for binary features.

the context of configurable software systems. The authors exclude a number of designs due to an infeasible number of measurements, and advocate the use of four designs, including the Plackett-Burman Design and Random Designs. Assuming a discrete domain of values for each numeric option, the former design requires each combination of levels for each pair of numeric options to occur equally. The latter design is advocated not least because of an negligible number of constraints among numeric options (Siegmund et al., 2015). For further reading on more detailed descriptions of experimental designs, refer to Antony (2014).

In conclusion, finding a suitable sampling strategy remains a tasks with many aspects

to consider. Depending on whether binary, numeric, or both types of configurations options are present, sampling strategies are selected, respectively. To derive mixed configurations, first, samples are selected from both binary and numeric configuration options. Second, the final sample of mixed configurations is computed as the cross-product of binary and numeric partial configurations. As stated by Siegmund et al. (2015), the treating binary and numeric options separately is justified since usually binary options enable or disable functionality while numeric options merely tune functionality.

In the context of out methodology, a suitable sampling strategy might include combining different strategies, either due to different types of configuration options, or due to multiple coverage criteria to meet. As a guideline for selecting a sampling strategy, especially in the context of performance assessment, no clear recommendation can be given. However, from samples selected via random sampling (Sarkar et al., 2015) and pair-wise sampling (Siegmund et al., 2015) have shown acceptable results in terms of accuracy.

4 Methodology: Version Assessment

In the last chapter, we have covered means to understand variability, synthesize variability models for a given software system, and to select sample sets of configurations. To enable the assessment of performance evolution for configurable software systems, the next step in our methodology takes into account the dimension of time. As software evolves, multiple versions, or called revisions, of a software system exist. In this chapter, we address the question of how we can assess performance for multiple versions. Moreover, we ask whether we can describe a configurable software systems' performance evolution history without exhaustively assessing all versions by selecting only a sample set of versions. As illustrated in the methodological road-map in Figure 4.1, we summarize these two questions with the task of *revision sampling*.

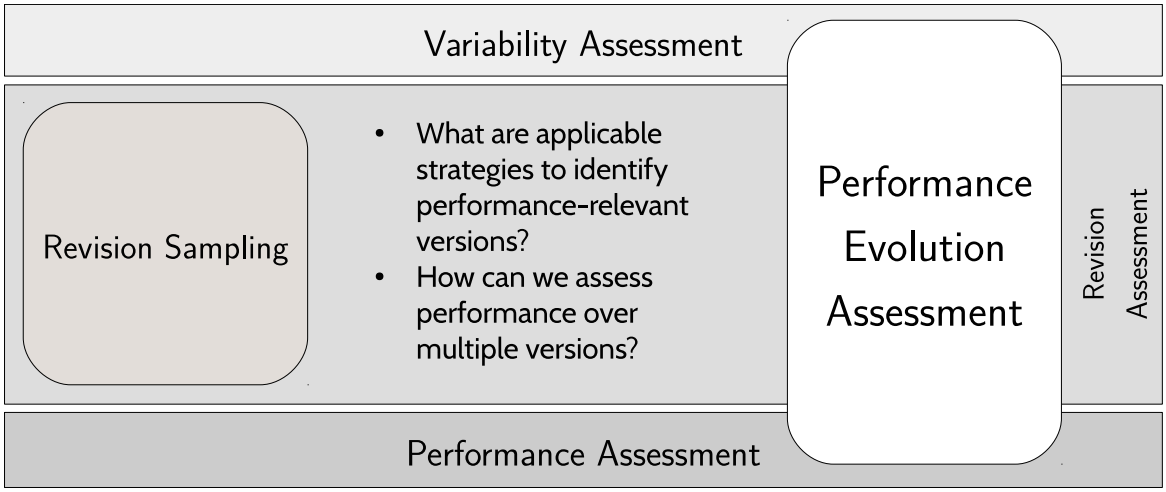


Fig. 4.1. Methodological road-map: questions to address with revision assessment.

The chapter is organized as follows. In section 4.1 we present the methodological requirements for a designing and selecting a revision sampling strategy. In section 4.2 we propose five approaches to revision sampling. In section 4.3 we evaluate the different approaches against exhaustive measurements of a selection of configurable software systems. Finally, in section 4.4 we conclude the chapter and discuss the approaches' applicability in the context of our methodology.

4.1 Towards Revision Sampling

Research so far has addressed the assessment of a software system's revision history under the umbrella of repository mining, for instance, to localize bugs (Moin and Khansari, 2010) or performance regression (Heger et al., 2013). Nonetheless, so far there exists little to no research addressing the question what the choice of versions might reveal about performance evolution. The task of selecting resources and a sample

set of versions to analyze can be conceived as a sampling strategy, where the objective is to cover interesting entities (performance changes, in our case) while trying to limit the sample size to keep the required effort reasonable. Before we present different approaches to select a sample set of revisions, we define general cornerstones for evaluating a sample set of revisions as well as respective revision sampling strategies.

Revision Sample Set. The first question is, what criteria we take as a basis for rating a sample set of revisions as *meaningful*. First, our intention is to obtain a representative description of a software system's performance change history while only assessing a fraction of revisions. That is, assessing a representative sample of revisions should yield a performance change history similar to the assessment of all revisions. Since we are interested in revisions for which performance measurements change, those revisions should be contained in a representative sample. Second, especially in the context of variability, a performance change might only affect a subset of the assessed configurations. A representative sample set, therefore, should describe a history of significant performance changes with respect to all variants assessed. We consider a performance change to be significant, if it affects a significant portion of the sample of variants (*effect range*) and if the relative change in performance measurements is significantly large (*effect magnitude*).

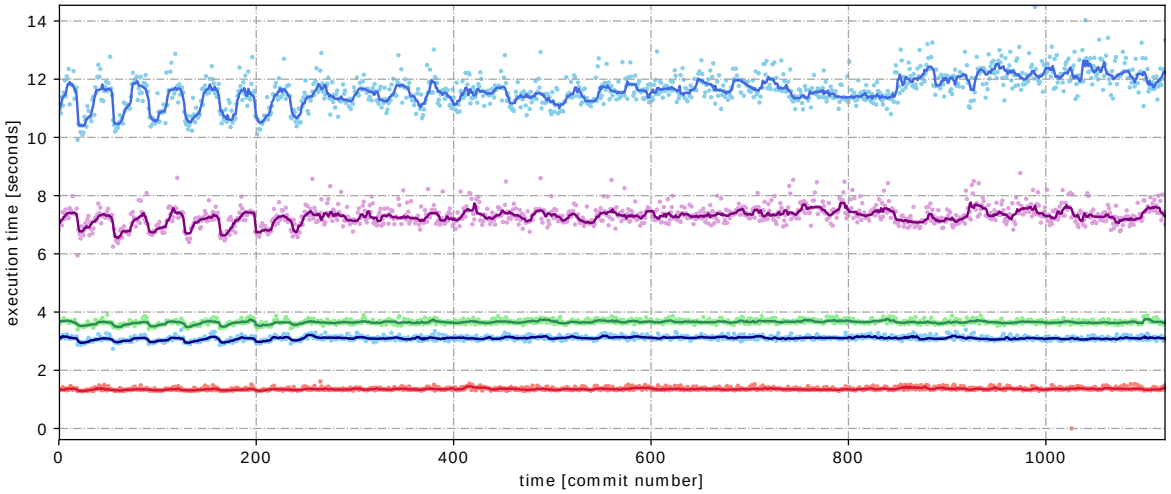


Fig. 4.2. Performance evolution history (execution time) five different variants of GNU XZ

To illustrate the two facets of performance changes, in Figure 4.2 we present a history of performance measurements (execution time in this case) for a small-scale configurable software system, file compression tool called *GNU XZ Utils*. The graphic illustrates execution time measures for executing a standard compression benchmark for 1,135 different versions and covers a version history of about nine years.
In this excerpt, we only show the execution time measures of five different variants, i.e., each version was assessed with five different configurations. Each performance measurement is depicted by a single marker. In addition, we provide a plot of a moving mean, to identify trends. We see that each variant has a certain performance level, and overall, the performance measurements remain stable. While the blue and purple variant are generally more volatile, for all variants, we see performance measurements

oscillating during the first 250 versions. With regard to the two facets of significance mentioned earlier, not all variants are affected similarly by this effect to the same degree.
While the former significance criterion can unambiguously defined by a threshold number of variants, for the latter one one needs to define how to summarize relative performance change among all variants. For instance, a performance change may have a significant magnitude, if the average deviation of performance measurements for all variants is greater than a specified threshold value.

Revision Sampling Strategies. The second question addresses the rationale behind a revision sampling strategy. To obtain representative sample sets, sampling strategies are intended to utilize knowledge about the volume to select sample sets from. For instance, pair-wise sampling aims to cover most feature interactions. Similarly, we demand for a meaningful revision sampling strategy to exhibit a certain rationale or coverage criterion. If we conceive a sampling strategy as a binary classificator that, in our case, decides whether in a revision a performance change is likely, or performance measurements might have changed compared to prior commits, we want this classifier to be sensitive, i.e., to have a preferably high true positive rate. That is, a sampling strategy is meaningful if we learn which revision features most likely indicate performance changes.

In conclusion, when designing a revision sampling strategy, we ask for a plausible rationale or coverage criterion with respect to performance changes. A resulting sample set of revisions, in addition, is representative, if the contained revisions sketch performance changes. For the context of this methodology, we remain with a clear definition of when performance changes are significant. Based on these assumptions, in the next section we propose a selection of five revision sampling approaches based on different observations.

4.2 Revision Sampling Strategies

As there are no established approaches to select sample sets of versions for configurable software systems, especially with respect to performance assessment, in the following five subsections, we propose five different strategies to address this problem. Our proposals utilize different sources of information regarding the software system studied, or are inspired by approaches that have shown to be applicable in different contexts. An overview and classification of the five proposals is presented in Figure 4.3. Our approaches can be grouped into two categories. The first category incorporates meta-data about a commit, including its lifetime (how long it has been the latest commit), and the number of lines of code modified by it. For the first category, sampling strategies intend to achieve high coverage of the overall version lifetime, or modifications, respectively. The proposals listed in the second category may as well take into account meta-data, yet for a different purpose. The classification-based proposals' use meta-data along with performance measurements to learn and predict the likelihood of a commit introducing performance changes. The remaining approach, in the overview referred to as performance history approximation, adapts the bisection approach by

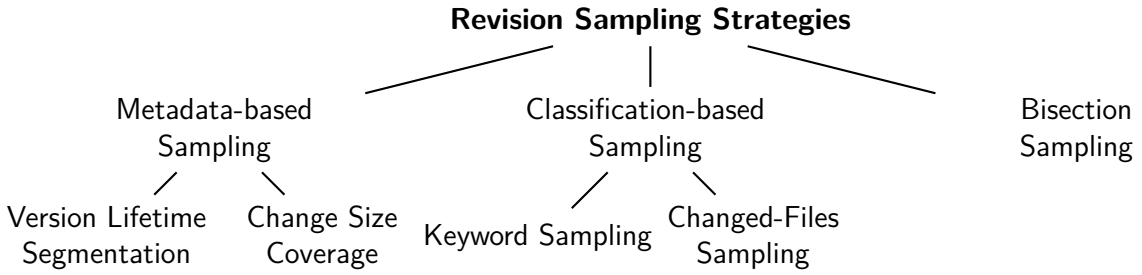


Fig. 4.3. Overview of our proposed revision sampling strategies.

Heger et al. (2013) mentioned earlier. The approach continuously expands a sample set until no performance-changing commits can be identified.

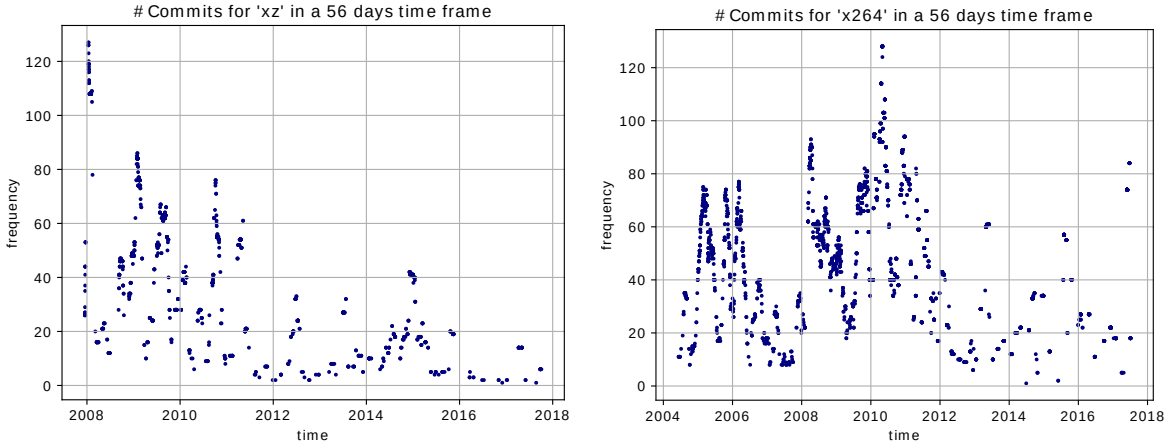
4.2.1 Keyword Sampling

The first version sampling strategy we present is driven by the conception that a commit message usually summarizes the changes of a commit in terms of what has been implemented, modified, removed, or what problems have been fixed. Commit messages can include keywords or phrases that indicate a performance context, such as "fixed performance bug ...", or exhibit information about the version of the software system, such as "bumped version number to ...''. Based on this information, we propose to use pattern matching to check whether a commit message contains a keyword that might indicate a performance context, or a new version, respectively. The sampling algorithm works as follows. Given a set of keywords, such as "performance, bug, fix, slower, faster, ...", we first derive the word stems for each keyword. Second, for each commit message, we split the message into separate tokens and derive their corresponding word stems. Third, we match all sets of resulting word stems (one set per commit message) against the set of keyword sets and retain those commit messages, for which sufficiently word stems are contained. As these commit messages contain our previously defined keywords, we select the corresponding commits as our version sample set.
This approach is simplistic and its accuracy clearly depends on the quality of the initial selection of the keyword set as well as the developer's discipline of documentation. Moreover, more sophisticated ways to compute similarity between texts, such as the tf-idf-value and various similarity metrics (Huang, 2008). However, we only propose this strategy to evaluate the overall applicability of approaches driven by text similarity to the problem of version sampling.

4.2.2 Version Lifetime Segmentation

The second approach to select sample versions from the history of versions is based on assumptions and observations obtained via repository mining. The following approach is driven by the assumption that performance changes are more likely to occur when the software system is revised frequently in a short period of time. Not only is this simply due to a higher number of revisions. We can also assume that if a software system has not been revised for a long period of time, changes in terms of performance-relevant fixes were not necessary, or have been deferred. Although postponing performance-relevant fixes has become sort of a virtue (Molyneaux, 2014), for the latter case there

can exist other reasons, including organizational constraints, or performance issues being undetected at that time. That is, we assume that those versions that have been the latest version for a long time due to the absence of changes as well as the necessity thereof can sketch a software systems performance history. We do not assume that long-lasting versions introduce performance changes, but they enable the segmentation of a software systems version history in a way that it represents a large portion of the software systems lifetime.

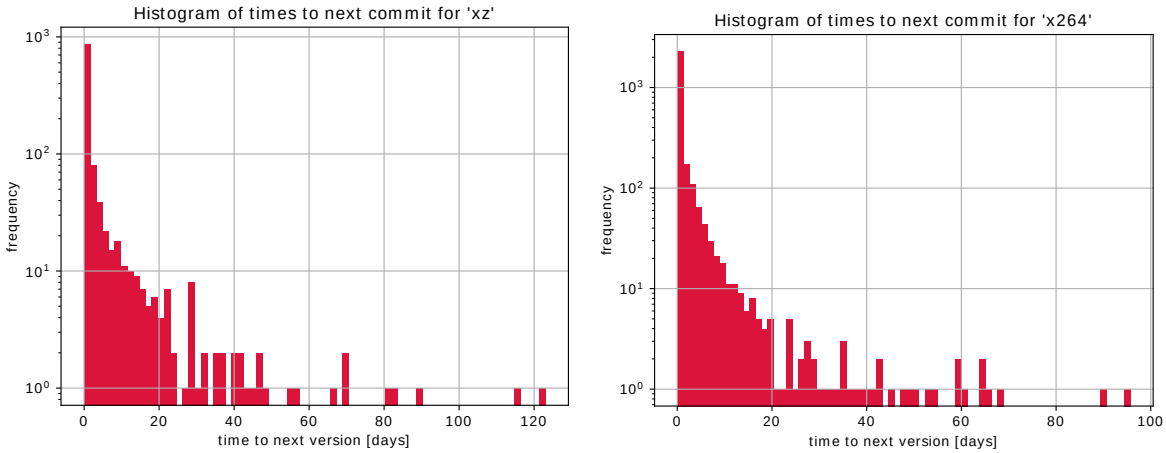


(a) Activity graph for `GNU XZ`, generated from 1,135 versions between December 8, 2007, and August 14, 2017.

(b) Activity graph for `x264`, generated from 2,851 versions between June 3, 2004, and June 26, 2017.

Fig. 4.4. Commit activity for two sample systems, the compression utility `GNU XZ` and the video encoder `x264`. For each version, the activity is measured as the number of commits that were pushed within a certain time-frame of eight weeks.

Making the connection with our methodological context and the aim to design a version sampling strategy, we intend to achieve a high "lifetime coverage" with a small number of versions. For this reason, we have evaluated the "lifetime" of versions of different software systems. In the following, lifetime of a version or commit refers to the period of time between a commit and its successor. We have investigated the distribution of version lifetime for two open-source software systems, a free file compression tool, *GNU XZ*, and a free video encoder, *x264*. – This selection is, by far, not representative, yet the observations obtained from systems document our assumptions. Since we will refer to those two and other systems for the evaluation, we answer how and why these systems were selected in the evaluation in chapter 6. – The first observation regarding the lifetime of single versions is illustrated in Figure 4.4 for the two software systems, respectively. The figure, for each version, shows the activity during development of both systems. For each commit, we have counted the number of commits preceding and succeeding it within a four week time frame, respectively. One can see that for both software systems we can identify spikes with a high commit frequency as well as plunges with little to no commit activity. Moreover, if we look at the histogram of all commits lifetime for both systems respectively as illustrated in Figure 4.5, we see that there actually are many commits with a short lifetime (activity spikes) as well as only very few commits with a long lifetime (plunges).



(a) Distribution of time to next commits for `GNU XZ`

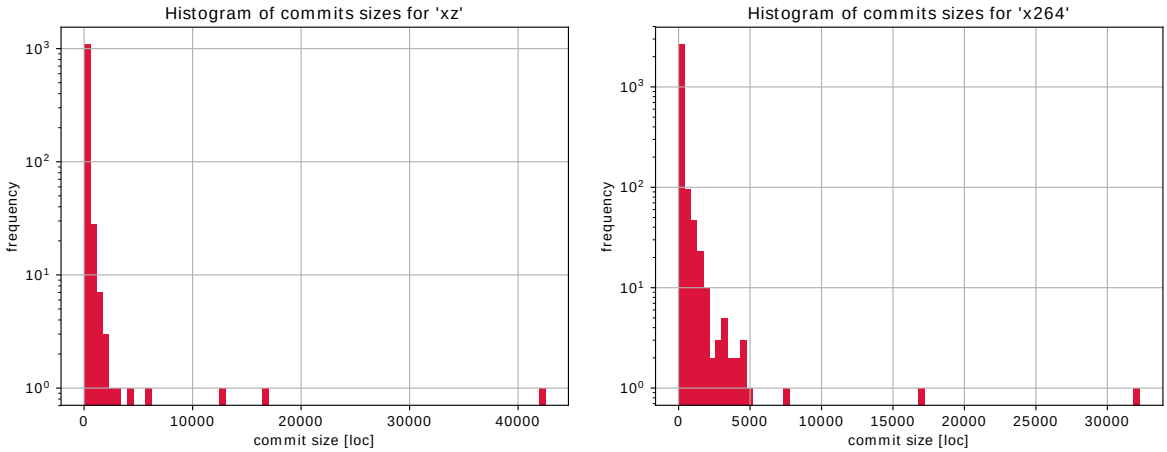
(b) Distribution of time to next commits for `x264`

Fig. 4.5. Distribution of time to next commits for two configurable software systems, measured as the distance between a commit and its successor.

Based on the aforementioned assumptions as well as the distribution of version lifetimes, when translating this in the context of a version sampling strategy, we can achieve high "lifetime coverage" by selecting very few versions from a list of versions sorted by lifetime in descending order. This sampling strategy picks the longest-lasting commits until a desired coverage threshold, or version count threshold number is reached. This sample selection of commits is a segmentation or clustering of the overall version history. We assume that each segment represents a (sort of) steady state of the software systems performance history.

4.2.3 Change Size Coverage

The third approach we propose adapts the idea of the previously mentioned version lifetime segmentation. The following sampling strategy design is driven by the assumption that a version or commit is more likely to introduce performance changes to a software system if it modifies a large portion of the code base. We assume that the effect of modifying a single line of code is not as significant as modifying multiple lines of code. Of course, modifications can accumulate, or trigger already existing interactions, yet from a black-box perspective, commits affecting more lines of code are more likely to introduce performance changes or to trigger interactions. That is, for designing a version sampling strategy, we aim to cover most of all changes made to the software system with a few number of versions. Similar to the assessment of version lifetime in Figures 4.4 and 4.5, we have investigated the distribution of commit sizes in terms of lines of code for the two software systems *GNU XZ* and *x264*. As illustrated in Figure 4.6, we see that, by far, most commits modify less than 2,000 lines of code, whereas very few commits modify or add larger code sections.
Based on these observations and given the assumption that larger commits are more likely to introduce performance changes, we propose to obtain a sample of versions by selecting versions from a list of versions sorted by commit size in a descending order. Similarly to version lifetime segmentation, one can specify a threshold of sample set



(a) Distribution of changed lines per commit for `GNU XZ`

(b) Distribution of changed lines per commit for `x264`

Fig. 4.6. Distribution of commit sizes in lines of code for two configurable software systems, `GNU XZ` and `x264`.

size or commit size coverage to achieve. We assume that a sample set of versions obtained using this approach is likely to cover significant performance changes since a larger portion of the overall change history is covered.

4.2.4 Bisection Sampling

The next sampling strategy that we propose is an adaption of the revision sampling approach used by Heger et al. (2013). For a software system's version history along with test cases and corresponding performance measurements, the authors aim to identify those commits which may have introduced performance regression. The authors have used a binary-search-like approach to continuously bisect the version history to find commits for which performance measurements deviate significantly from previously measured ones. Once a relevant commit is identified, it can be subject of further root-cause analysis.

We adapt this approach to extract performance-relevant commits given an initial sample of versions along with performance measurements. A performance-relevant commit in this context is a commit for which performance measurements significantly deviate from measurements of preceding commits. The sampling strategy is applied as follows. Given an initial sample of n versions and corresponding performance measurements, the version history is segmented into $n-1$ segments. Each segment is clearly specified by two commits. For each segment s_i, the algorithm computes the difference in performance measures δ_i for the start and the end commit. The segment with the largest deviation then is bisected. For the resulting two segments δ_{i_1} and δ_{i_2}, the performance measurement differences are calculated. If the absolute difference between δ_{i_1} and δ_{i_2} is greater than zero, one of the two children segments is "steeper", i.e., has a greater performance measurement difference than the parent segment as exemplified in Figure 4.7.

One can specify a minimum difference threshold to retain those resulting segments. This procedure is repeated until no further segments can be bisected without violating

the minimum performance difference threshold. That is, the algorithm approximates an interpolation of the overall performance history while focusing on sketching the steepest changes. In addition to selecting an initial sample randomly, we also propose to use version lifetime segmentation sampling to obtain segmentation of the version history that segments it with respect to development activity.

4.2.5 Changed-Files Sampling

The last version sampling strategy that we are going to present is a binary classifier that predicts whether a commit is likely to introduce performance changes. The idea behind this classifier is that performance changes might depend on changes in specific code sections or combinations thereof. For each commit we can retrieve which code sections, or files have been modified. If there exists a relation between a commit's file change set and performance changes, we could learn it via machine learning and consequently predict the performance change likelihood for arbitrary commits.

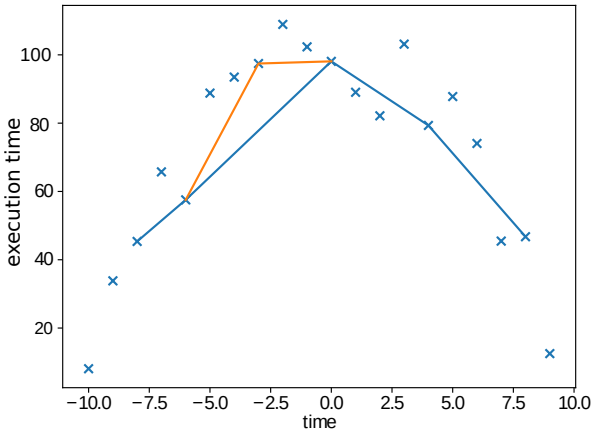


Fig. 4.7. Given a initial sample of five versions and four segments (blue line), the steepest segment is bisected and replaced by two segments (orange). The first of the orange segments is steeper than the segment that was bisected.

We propose a sampling strategy that is based on a initial training selection of commits to learn a possible relation between a commit's set of changed files and performance changes. For each commit in the initial learning set, the performance difference to its predecessor is calculated. Since we are interested in any change in performance, not just performance degradation, we consider the absolute performance difference as the performance change introduced by that commit. Next, we are using classification and regression trees (CARTs) to approximate a function that maps a commit's set of changed files to its absolute performance change. If the classificator is accurate enough, it can be used to rank the commits that are most likely to introduce performance changes. The sampling strategy can be configured with two parameters, a, the size of the set to learn performance change likelihood from, and b, the actual sample size to return. While the first a parameter determines the number of performance measurements required by this strategy, the second one, b, determines how many top-ranked revisions are returned by the sampling strategy, and therefore the actual sample size.

Moreover, for this strategy, we can modify the way to select a training sample. The simplest solution is to randomly select a commits and assess performance for them and their predecessors. One could also select an initial learning sample based on heuristics to increase its informative value. For instance, since we intend to learn possible relations between a commits file change footprint and performance changes, we could apply the

commit change size sampling strategy presented in section 4.2.3 to possibly increase the informative value of our training sample by covering a large portion of the version history's overall change.

4.3 Strategy Evaluation

In this subsection, we evaluate and compare the revision sampling strategies described above with respect to their accuracy to approximate the overall performance history of a software system. First, we describe how we measure accuracy for a given revision sample; second, we present our corpus of systems we assess performance evolution for, and finally, we present and discuss the different sampling strategies' results.

4.3.1 Evaluation Setup

An accurate approximation of an overall performance history in our context is a selection of revisions along with its performance measures, which accurately sketches the performance history obtained from assessing all versions. From an accurate approximation one should be able to identify the same trends as from the overall performance history. We measure the accuracy of a given revision sample as follows. First, for the revision sample, we interpolate missing performance measurements based on the measurements for our sample. Second, we smooth the overall performance history, i.e., the performance measurements of all revisions, using a moving mean in order to sketch overall performance trends. Third, we compare our smoothed performance history and our interpolated approximation using the Pearson correlation coefficient. The more similar our approximated curve is to the overall performance history, the greater the correlation coefficient is.

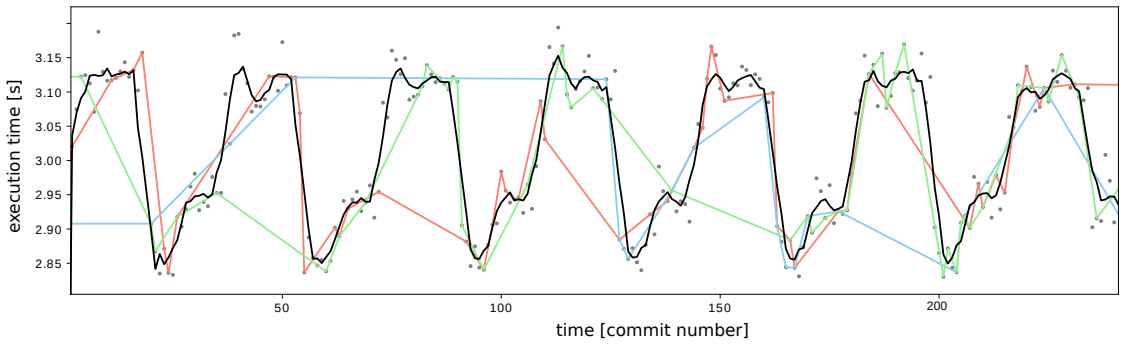


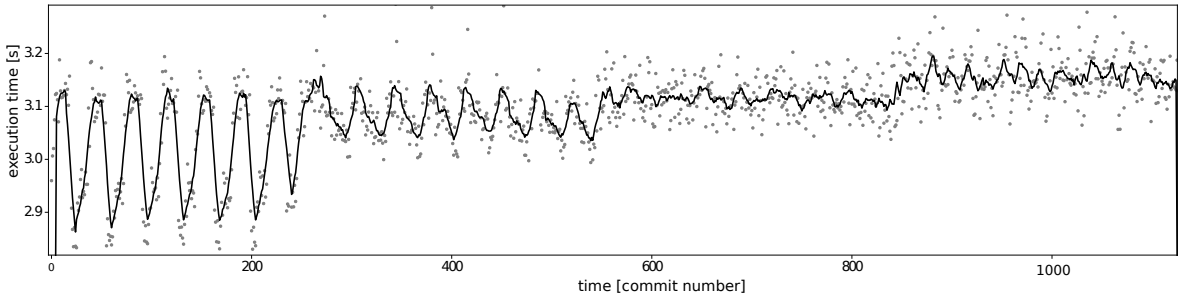
Fig. 4.8. Performance history approximations with different accuracy results.

In Figure 4.8, we illustrate performance history approximations with different accuracy. The grey scatter plot depicts the average performance measurements per revision, and the black curve represents the moving median thereof. The green, blue, and orange curves are respective approximations. One can see that the green and orange approximation are a more accurate approximation than the blue one since they more closely follow the overall performance history.
We evaluated all five revision sampling strategies for two different configurable software systems. The first system, GNU XZ, is an open-source file compression tool that is widely used among the open-source community. It exhibits a number of configurations

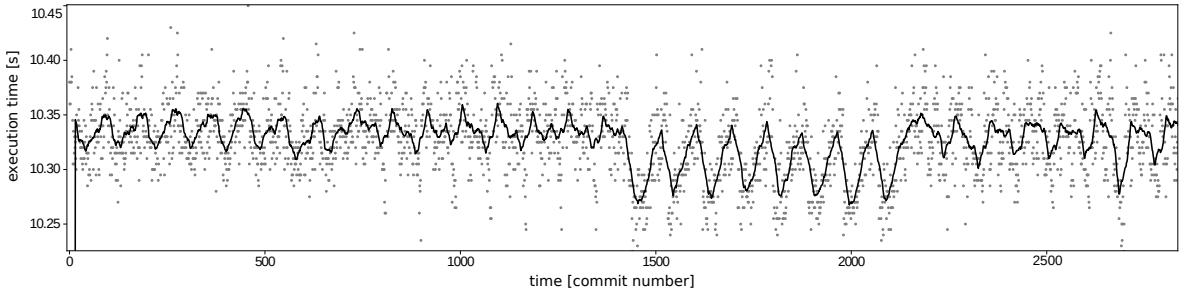
options via run-time parameters from which we derived 36 variants using pair-wise sampling. The second system, x264 is a free implementation of, and encoder for the H.264 video compression standard, commonly known as MP4. Similarly to GNU XZ, it can be configured using a wide range of run-time parameters from which we derived eight variants using pair-wise sampling. A detailed description of the systems' configuration options and how we derived the feature model is presented in chapter 6. For both systems, we obtained a clone of their respective git repositories and, as far as possible, compiled each version: 1,135 for GNU XZ and 2,851 for x264. For GNU XZ, we selected a standard file compression benchmark, the Canterbury corpus (2.8 MB), for GNU XZ since it contains different kinds of data (binary as well as plain text). For x264, we selected an uncompressed movie clip of 79.9 MB as a benchmark. All performance measurements (execution time) were conducted on a Linux machine (Debian 8) with a Intel Xeon E5-2680 v2 with 2.8 GHz and 16 GB RAM. For each version and variant, we repeated the experiment five times and selected the median of the measurements to exclude extreme measurements.

4.3.2 Result Description

For both assessed systems, we first present the overall performance history to give an impression of its overall shape. In Figure 4.9, for both systems, we show the average performance measurements per revision, calculated as the arithmetic mean over all 36 and eight variants, respectively.



(a) Average performance history for GNU XZ



(b) Average performance history for x264

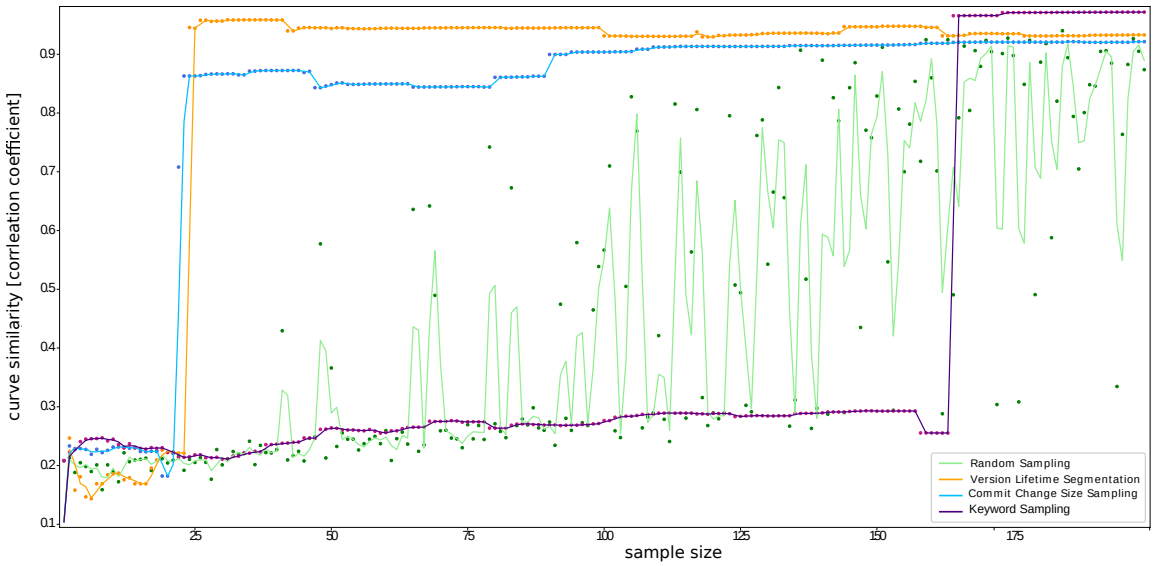
Fig. 4.9. Average performance history per version for both GNU XZ and x264

For GNU XZ, in the beginning, the performance measurements fluctuate significantly with an amplitude of up to 0.4 seconds, yet the amplitude decreases over time. Moreover, the overall trend of the performance measurements indicates an increase in execution time. This overall trend along with the decreasing amplitude of fluctuations

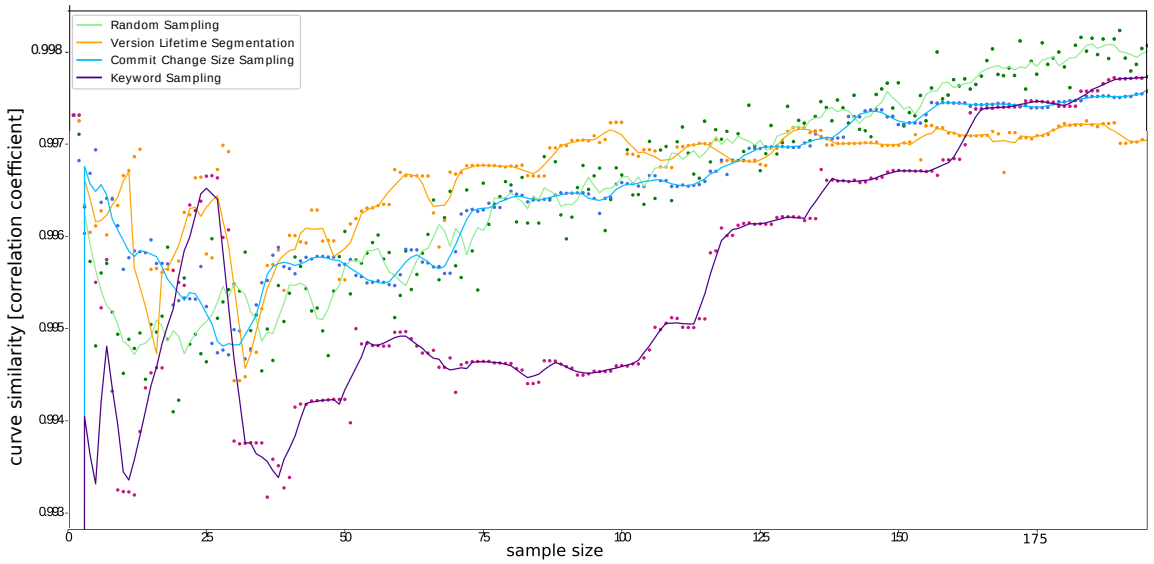
might indicate a software architecture that is getting more stable over time, yet becomes slower over time as well, which is in line with the theory of technical debt mentioned earlier (Guo et al., 2011). However, for x264, we could not identify such an overall trend as the performance measures remain stable for most of the version history. Only for the time span between commit 1,400 and 2,100, performance measures fluctuate around a smaller level of execution time. In conclusion, both systems' performance histories exhibit local fluctuations, yet the fluctuation range for x264 remains stable throughout the entire version history while for GNU XZ, the fluctuation range decreases as the software evolves. Moreover, for GNU XZ, the frequency of fluctuations, i.e., the number of fluctuations in a time frame, decreases over time, while it almost remains stable for x264.

Accuracy Evaluation. We were able to accurately approximate the performance history curve for GNU XZ with different revision sampling strategies. Our first comparison comprises keyword sampling, version lifetime segmentation, and change size coverage sampling. In addition, we compared those three sampling strategies to a baseline random sampling approach, whereby n arbitrary revisions have been selected. While the other three sampling strategies are deterministic, we repeated the random sampling a hundred times per revision to minimize variance. For the comparison, we sweeped the desired sample size for each software system from one to the total number of commits, respectively.
In Figure 4.10, we illustrate the accuracy measurements for four different sampling strategies. For GNU XZ, keyword sampling and random sampling performed relatively poorly. Compared to version lifetime sampling and commit change size sampling, they did not achieve an acceptable level of accuracy. For x264, version lifetime sampling and commit change size sampling did not perform significantly worse than the respective others. In general, for x264, we achieved far better accuracy measurements for all applied sampling strategies as accuracy measurements did not fall below 0.9.

Changed-files Sampling. For changed-files sampling, we chose a different type of visualization since the sampling strategy works with two parameters, the initial learning sample size, and the desired sample size. We visualize the accuracy as a heat map, whereby combinations of initial learning sample size and desired sample size are coordinates, and the color at those coordinates represents the corresponding accuracy measurements. Moreover, we let the sampling strategy operate in two different modes. First, we selected the initial learning sample randomly, and second, we selected the change size coverage sampling. We illustrate the accuracy measurements for both modes in Figure 4.11 and 4.12, respectively.
For the experiments with randomly selected initial learning samples in Figure 4.11, for GNU XZ, the accuracy increases for greater sample sizes. Moreover, the spread of accuracy decreases for greater training sample sizes. That is, when learning the relation of changed files and performance changes from a larger selection of revisions, fewer revisions are required for the actual sample to achieve a certain accuracy. For the experiments, where the initial learning samples were selected using the commit change size sampling, as illustrated in Figure 4.12, we see that the general level of accuracy is significantly higher. For instance, for GNU XZ, the accuracy ranges from 0.6 up to almost 1.0. In contrast to the previous mode, even a small training sample of 100 to



(a) GNU XZ

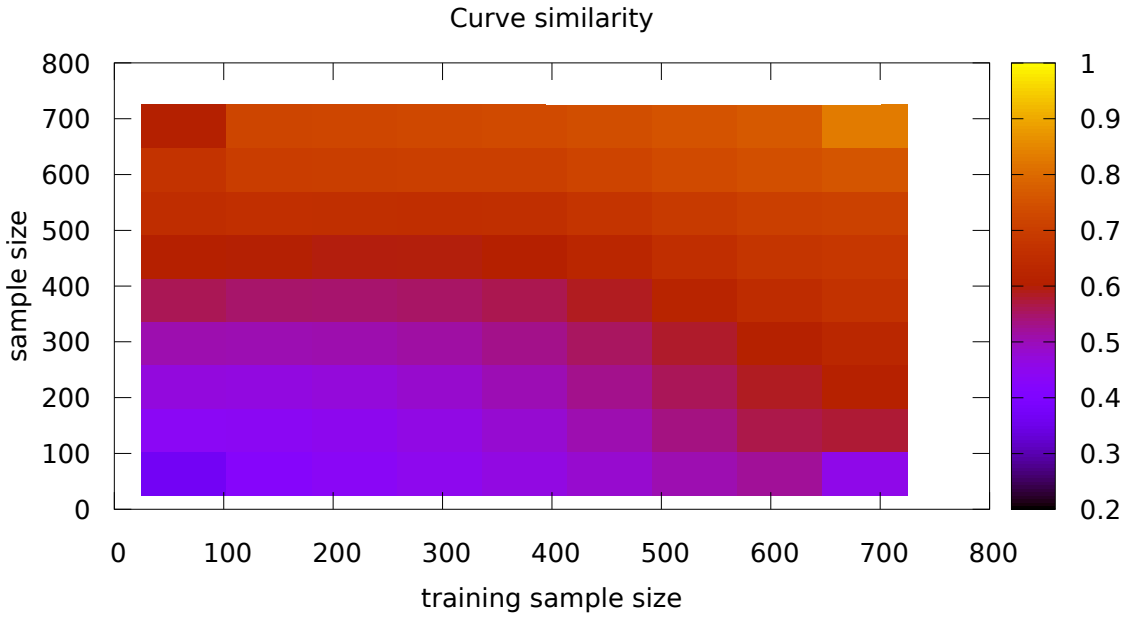


(b) x264

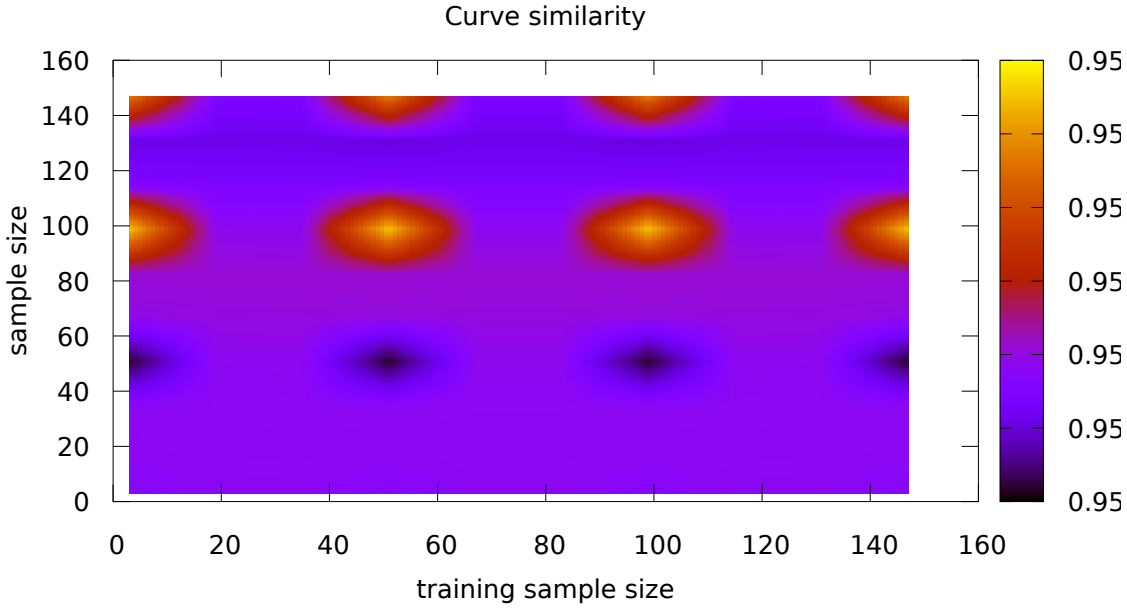
Fig. 4.10. Accuracy of four different sampling strategies: version lifetime sampling, random sampling, commit change size sampling, and keyword sampling.

200 revisions can be sufficient to achieve reasonable accuracy. Moreover, for a small sample size, but a great training sample size, the accuracy degrades.
Following the results for the previous sampling strategies, for x264, we also achieved high accuracy measurements for both changed-files sampling with random sampling as well as commit change size sampling. However, we did not measure any significant influence of either segmentation technique and accuracy did almost never vary for any combination of parameters.

Bisection Sampling. For the last sampling strategy, we continue using the same visualization technique as for changed-files sampling. Again, this sampling strategy



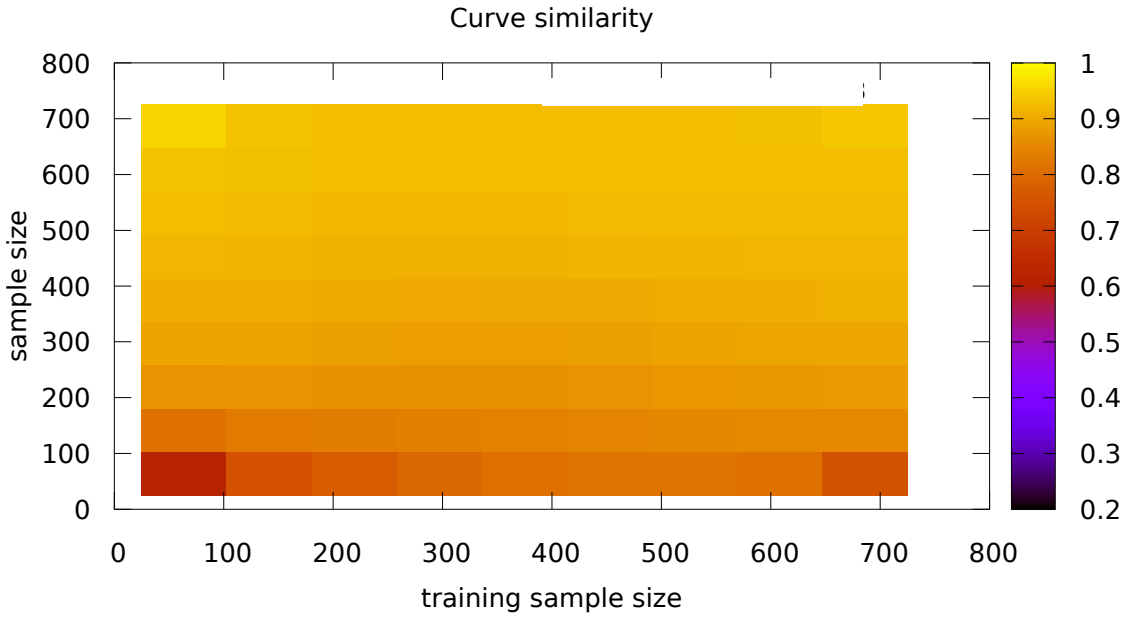
(a) GNU XZ



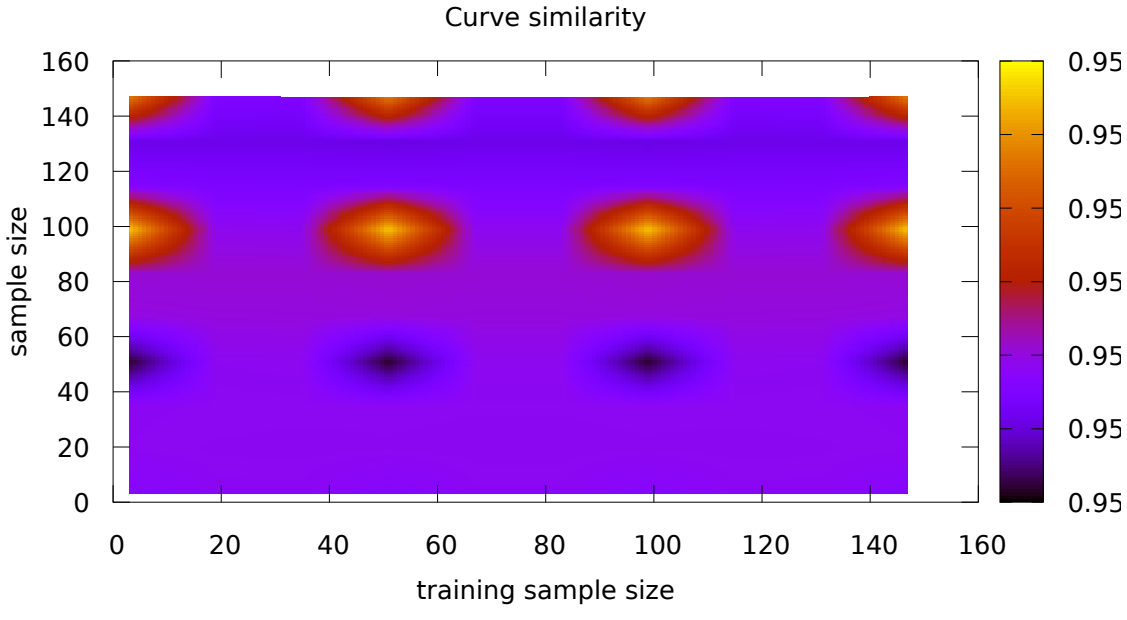
(b) x264

Fig. 4.11. Accuracy of changed-files sampling with randomly selected initial learning sample for GNU XZ and x264

requires two parameters, an initial sample size a, and a number of extension steps b to perform. The initial sample of size a can be selected randomly, or using version lifetime sampling. In Figures 4.13 and 4.14 we illustrate the accuracy measurements for both modes, respectively. We see that for the first operation mode in Figure 4.13 with random sampling, the overall accuracy is increases with a greater size of the initial sample size, which is in line with our observations for pure random sampling. The number of extension steps, however, does not exhibit a significant influence on the accuracy, yet only for a small initial sample size, where accuracy spread is high. Contrary to the first



(a) GNU XZ

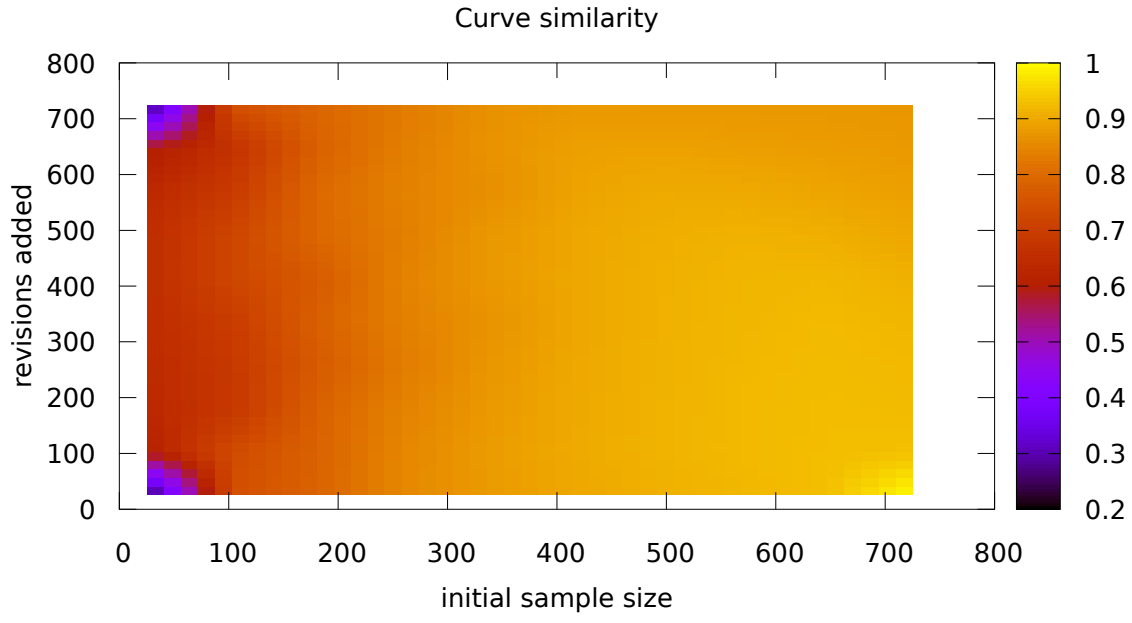


(b) x264

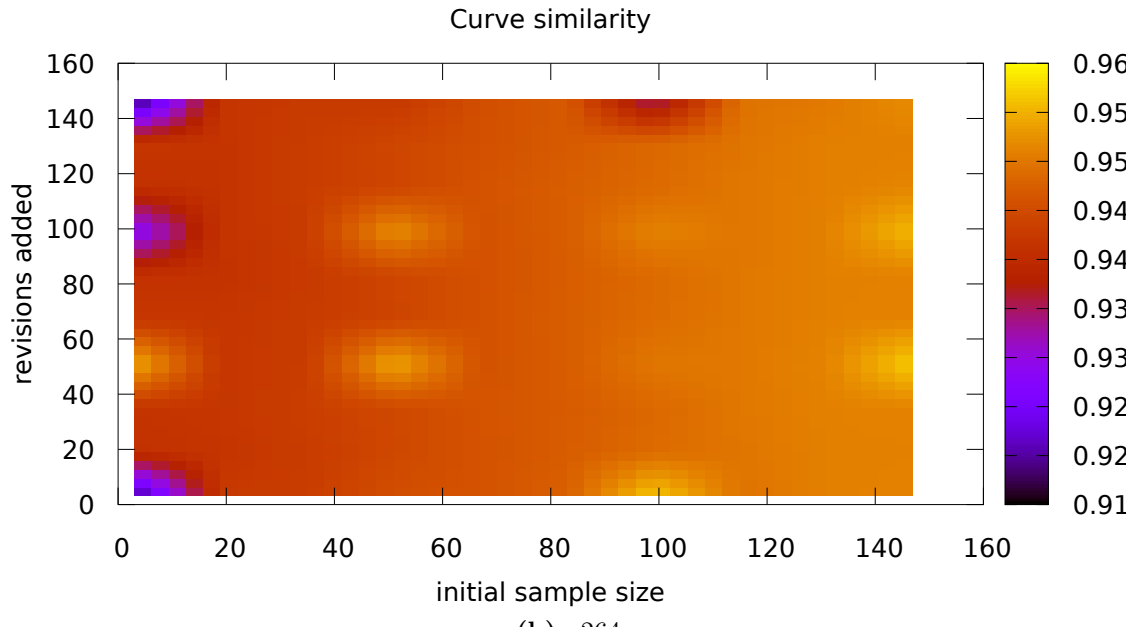
Fig. 4.12. Accuracy of changed-files sampling with the initial learning sample selected using commit change size sampling; for GNU XZ and x264

operation mode, in Figure 4.14, we illustrate the accuracy measurements for the second operation mode, where the initial sample is obtained via version lifetime segmentation. While most of the description of the previous operation mode also applies to this one, the operation mode in general performs better, as the minimum accuracy measurement is about 0.9.

Similar to the accuracy measurements for changed-files sampling for x264, we achieve high accuracy measurements for bisection sampling for both combinations with random sampling as well as version lifetime segmentation, whereby the latter combination



(a) GNU XZ

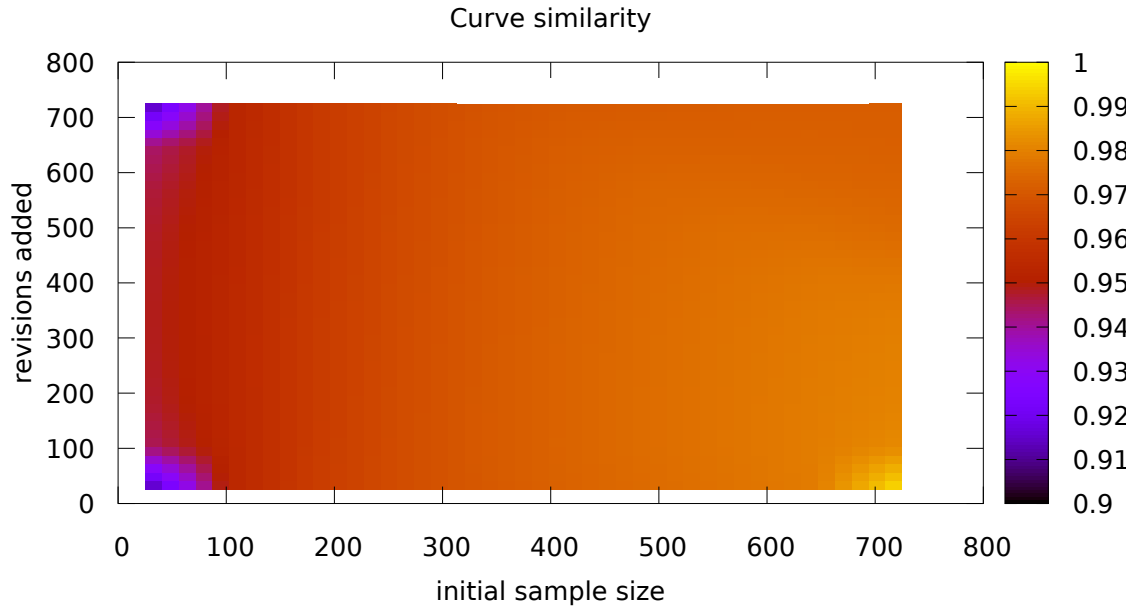


(b) x264

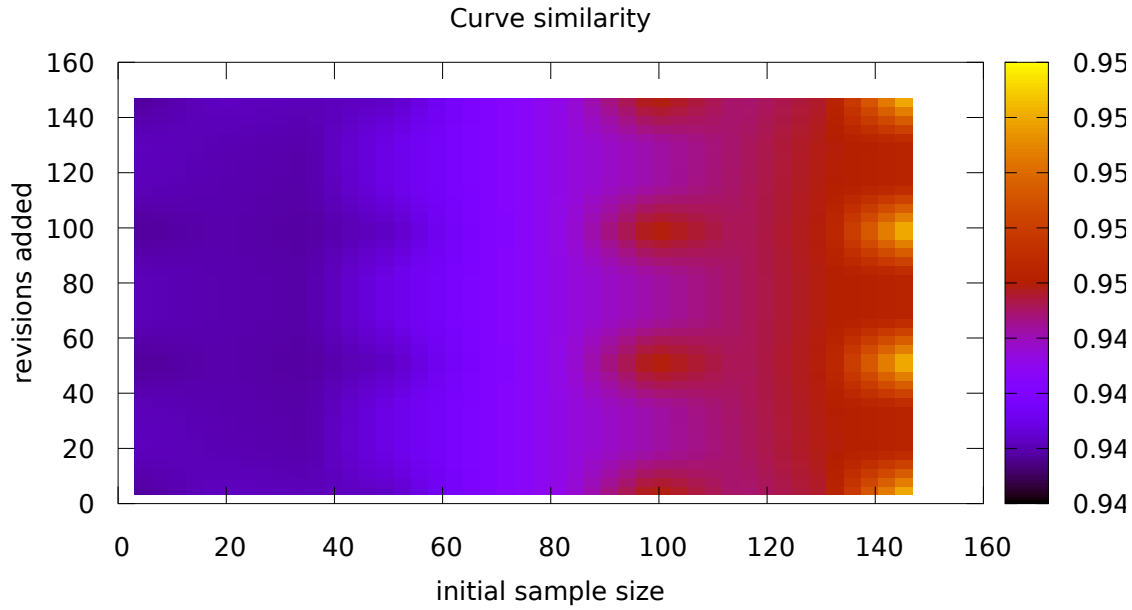
Fig. 4.13. Accuracy of bisection sampling with randomly selected initial samples for GNU XZ and x264

performed slightly better. However, the range of spread for accuracy measures for both operation modes is negligibly small.

Discussion. The accuracy measurements of the five sampling strategies presented above differ significantly between the two systems studied. While the results for GNU XZ are heterogeneous enough to suggest a sampling strategy over the others, for x264, following the results, every sampling strategy tested performed well. Furthermore, all accuracy measurements for x264 seem to remain almost stable for most combina-



(a) GNU XZ



(b) x264

Fig. 4.14. Accuracy of bisection sampling with initial samples selected via version lifetime segmentation for GNU XZ and x264

tions of input parameters. That is, to understand the discrepancy between both two systems, we re-evaluated our definition of accuracy. All accuracy measurements were calculated as the correlation coefficient between the average performance of all variants assessed and the interpolation obtained from a subset of the average performance history. Moreover, for the overall average performance, we consider a moving mean in order to smooth the performance history curve and sketch mid- to long-term performance evolution trends. While we obtained accuracy results of plausible shape for GNU XZ, for x264, the previously described accuracy measurements appear unlikely.

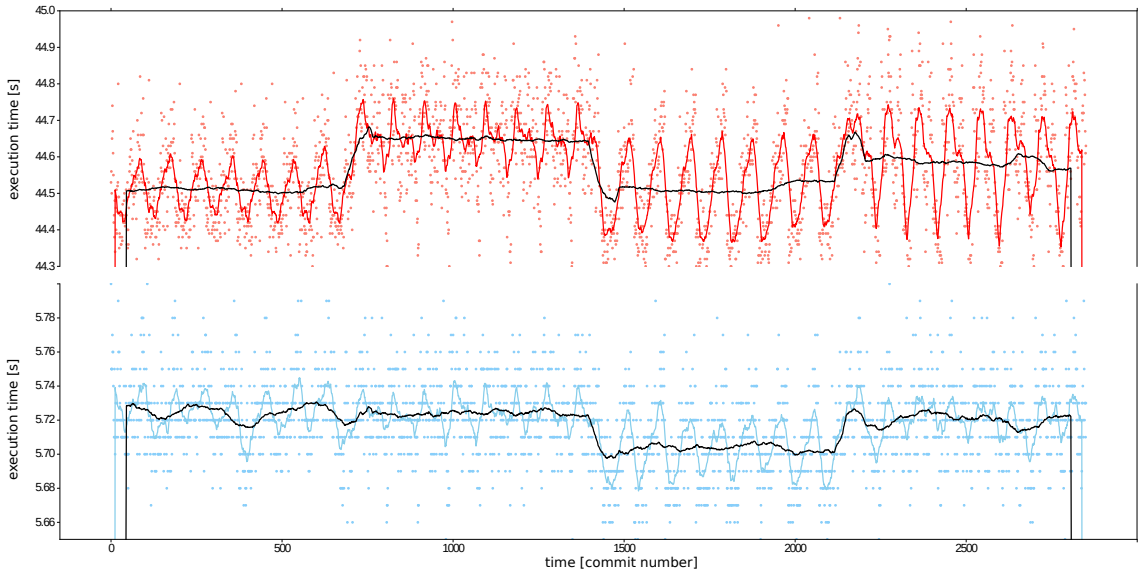


Fig. 4.15. Performance history for x264: worst-performing variant (top) and best-performing variant (bottom)

We identified two possible causes for those results. First, the smoothing factor, i.e., the frame size for the moving mean, influences how similar the smoothed curve looks to a straight line. If a greater smoothing factor is selected, the performance history curve can easily be approximated with an interpolation of only two points. Second, the performance history itself can be shaped quasi-linear. For the latter case, either the performance remains stable while the software system evolves, or the average performance history does not exhibit existing performance changes, for instance, when performance changes only affect few variants, or performance only fluctuates within a small range. From comparing the average performance history (cf. Figure 4.9) and the best- and worst-performing variant (see Figure 4.15) for x264, we learn that at around commit 650 performance significantly increases for the best-performing variant, while almost no effect is visible for the best-performing one as well as for the average performance history. Moreover, the spread range for performance measurements was not more than 0.1 seconds for the best-performing variant, and 0.15 seconds for the average performance history. Considering the range of 0.4 to 0.5 seconds for the worst-performing variant, the differently shaped performance curves suggest that the used evaluation technique is not necessary suitable to evaluate performance history, if effects are of too small magnitude or effect range.

4.4 Methodological Remarks

In the last subsection we have presented the evaluation results for our revision sampling strategies proposed earlier. Despite different accuracy measurements, we now put the strategies and their cost-efficiency in the context of our methodology. While a high accuracy is desirable, the revision sampling strategies proposed differ in the required number of performance measurements For instance, keyword sampling as well as version lifetime segmentation and commit change size sampling require no revision to be

Sampling Strategy	Parameters	Required measurements	Resulting sample size
Keyword Sampling	Sample Size N	0	N
Version Lifetime Segmentation	Sample Size N	0	N
Commit Change Size Sampling	Sample Size N	0	N
Changed-files Sampling	Sample Size N Learning Sample Size M	$2 \times M$	$M + N$
Bisection Sampling	Sample Size N Initial Sample Size M	$2 \times M$	$M + N$

Tab. 4.1. Overview of different sampling strategies, required parameters, performance measurements, and resulting sample sizes

assessed at all, while for changed-files sampling and bisection sampling the number of revisions that are necessary to assess exceeds the resulting sample size. In Table 4.1 we present an overview of the five proposed sampling strategies along with their parameter description, number of required performance measurements, i.e., revisions to assess, and the resulting sample size depending on the strategies parameters.
With respect to accuracy, from the evaluation we have learned that commit change size sampling, version lifetime segmentation, changed-files sampling with commit change size sampling, and bisection sampling with version lifetime segmentation performed best. Commit change size sampling does not require any performance measurements. We consider the cost of collecting information about all commits as well as ordering them with respect to commit change or lifetime coverage as constant since the time required only depends on the host machine used for sampling, and it only required a few seconds for our evaluation setup.
Next, changed-files sampling with commit change size sampling has shown acceptable accuracy, yet only for relatively great sample sizes compared to pure commit change size sampling. Moreover, the selection of a suitable learning sample size does not seem trivial, as accuracy increased up to a certain learning sample size, yet decreased for greater sample sizes. That is, it is hard to estimate a sweet spot size for a learning sample that on the one hand contains enough knowledge to learn a possible relation between files changed an performance changes on the one hand, and on the other hand does not contain too many revisions diluting the learning sample's informative value.
Finally, bisection sampling with version lifetime segmentation has shown high accuracy for almost all parameter combinations, yet compared to the previous sampling strategies, the required number of revisions to assess is significantly higher. Contrary to changed-files sampling that also utilizes an initial sample for which twice the number of measurements is required, this sampling strategy additionally requires two performance requirements for each segmentation step.

In conclusion, based on our case study we advocate the use of commit change size version lifetime segmentation sampling as the ratio of required performance measurements and resulting accuracy is optimal. However, note that the case study corpus only contains two software systems and that our results are far from being representative. However, based on our observations, we were not able to disprove the assumption of a relation between a commit's change size and its potential influence on performance.

5 Methodology: Performance Assessment

The last two chapters covered methodological guidelines for variability as well as version assessment. While those two terms, *variability* and *versions* represent two orthogonal dimensions of a configurable software systems' evolution history. To provide a closed description of guidelines for performance evolution assessment, this chapter finally presents a guideline to assessing performance for a given variant and version. In Figure 5.1, we present the methodological road map and questions to be answered in this chapter.

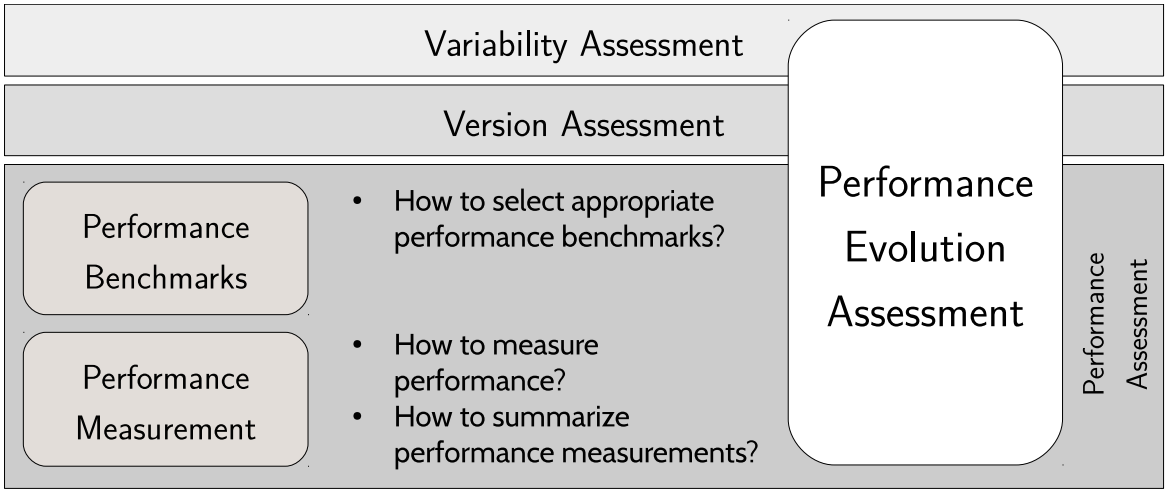


Fig. 5.1. Methodological road-map: performance assessment

First, in section we categorize software systems with respect to the availability of suitable performance benchmarks; second, we outline the general properties of profiling tool support used throughout dynamic program analysis; finally, we discuss different statistical with respect to their applicability in the context of performance assessment and robustness.

5.1 Performance Benchmarks

The essential part of assessing performance for a software system is the choice of a benchmark that one wants to evaluate the software against. For the choice of a suitable benchmark, two consecutive questions need to be answered. First of all, we need to specify, what aspects of performance we intend to evaluate for our software system. As presented earlier in chapter 2, the term *performance* is generic as it is commonly outlined by different key performance indicators. For instance, for a web

shop application, good performance is characterized by low response time and high availability, while for file compression, one might rather conceive high throughput as good performance. That is, given a specific aspect of performance, i.e., one or more KPIs, for a software system, from testing against a suitable performance benchmark one needs to be able to judge whether the specified performance goals are met or not. Next, once we have specified performance indicators fitting our software system, we need to select a benchmark, i.e., a repeatable task for the software system for which we can evaluate performance. To keep performance measurements comparable throughout all versions and variants, it is required to use only one benchmark per assessment. Following this principle, and to even compare different software systems, practitioners are advised to refer to reusable, or more general performance benchmarks whenever possible. To illustrate this, we can divide software systems into three general categories for which we are presenting separate strategies.

First, in the the easiest case, a software system provides software tests, or performance benchmarks, to use for performance assessment. The idea of using an existing test suite to assess performance is straightforward and has already been applied to detect performance regression (Foo et al., 2010; Heger et al., 2013).

Second, a software system can offer functionality for a domain, where standardized benchmarks have been established, or are commonly used to compare performance measurements. For instance, for the domain of file compression there exists a long tradition of using standardized file sets as performance benchmarks, such as the Canterbury or Calgary corpus[1]; for video encoding, the xiph.org[2] foundation provides a large set of video benchmark files; and, more general, for processors the Standard Performance Evaluation Corporation provides (SPEC) standardized benchmarks for floating point operations. Whenever a software system falls under a domain for which standardized benchmarks exist, we advocate to use those.

Finally, if neither a performance benchmark or test suite is available for a software system, nor domain-specific benchmarks exist, the task of designing a benchmark is left to the practitioner evaluating the software. As the conception of performance is generic and highly context-dependent, there is no standardized recipe for designing a performance benchmark. However, the main criteria to be satisfied include *expressiveness*, *cost-efficiency* and *reproducibility*. First, a suitable performance benchmark should clearly express what a desirable and unfavorable performance measurement is, given a specified key performance indicator. For instance, if we evaluate performance in terms of response, or execution time, minimal measurements are desirable. Second, effort that is required to evaluate a software with a benchmark must be reasonable. Since performance measurements are usually repeated to decrease measurement bias, or used for different variants and revisions, performance benchmarks should be limited in terms of size. However, a performance benchmark needs to be large enough to sketch performance changes or deviations. Finally, the construction of a benchmark must be reproducible, i.e., transparent and plausibly designed as the overall value of a performance benchmark depends on whether we can draw any conclusion from performance measurements obtained from it.

In conclusion, for the context of our methodology, the advocated guideline for selecting and designing a performance benchmark for a configurable software system is to select

[1] See `http://corpus.canterbury.ac.nz/descriptions/` for a detailed description.

[2] `https://media.xiph.org/`

a benchmark as generic as possible. If there is no reusable benchmark available, the design of any performance benchmark should follow the criteria of expressiveness, cost-efficiency, and reproducibility.

5.2 Profiling

With a software system along with a benchmark set up, the only choice left before assessing performance is to select a means to obtain and collect performance measurements. For this type of tool support for dynamic analysis, commonly referred to as profilers, exists a variety of solutions, varying in functionality and scope of application. Gosain and Sharma (2015) have surveyed a selection of profilers for dynamic analysis. The authors categorize their selection into three categories, including VM-profiling-based, instumentation-based, and profilers based on aspect-oriented programming (AOP). While for VM-based profilers, make use of profiling functionality offered by the respective platform, such as the Common Language Interface (CLI) for the family of .NET languages, or the Java Virtual Machine (JVM), instrumentation-based profilers add instructions to the software to the software's code base at compilation using instrumentation preprocessors, or to the software's executables. AOP-based profilers accommodate profiling functionality in separate modules called aspects. An aspect's code, called advise is executed whenever a corresponding event, called join point, occurs, for instance a call of a function that one would like to measure performance for. Several join points can be summarized by a set of join points, called point cut. While VM-based profilers are limited to a family of programming languages, and, as well as instrumentation-based, usually introduce a run-time overhead that might dilute performance measurements. In contrast to that, AOP-based profilers usually perform better, yet are limited to the code base as aspects are woven into the code base during re-compilation. Moreover, they require a higher level of expertise to design and deploy a performance test setup compared to the former two categories (Gosain and Sharma, 2015). In addition to the aforementioned profiler categories, statistical profilers such as *Performance Counters for Linux*[3], or *perf*, monitor a software system performance, for instance hardware counters for regular intervals, and therefore provide an approximation of the software system's performance with less overhead than instrumentation-based profilers.
In the context of our methodology, we are advocating the use of the Unix command `time`[4] to measure execution time along with a number of further performance metrics for a software system. The command is contained in most Linux distributions and, from a black-box perspective, provides sufficient functionality to sketch performance in terms of metrics for execution time, memory consumption, or I/O statistics. While tools falling under the aforementioned categories are powerful tools for dynamic analysis and program optimization, `time` provides a basic, yet exhaustive support for assessing performance for software system run on a single host machine.

[3] See `http://man7.org/linux/man-pages/man1/perf-stat.1.html` for further information on `perf`.
[4] See `http://man7.org/linux/man-pages/man1/time.1.html` for the man page of `time` and provided performance metrics to record.

5.3 Statistical Considerations

In this section, we now discuss the appropriate statistical means to summarize, compare and interpret measurement results. For the remainder of this section, we will refer to the following two scenarios. First, to obtain robust results, the assessment of a single variant needs to be repeated multiple times. Consequently, to report a single result per metric, the measurements for a single variant need to be summarized. Second, the assessment of performance for a variant may comprise several use cases, for instance file compression and decompression for a compression software. Therefore, performance measures aggregated from different benchmarks need to be summarized accurately.

5.3.1 Measures of Central Tendency

For each test run of a software system or variant, we obtain a single-valued measurement per performance metric. Since we repeat each test run n times per variant, we obtain a data record X with n measurements $X = X_1, X_2, \ldots, X_{n-1}, X_n$. While the arithmetic mean is commonly considered the right way to summarize data records and report a representative average value, we need to be more cautious with how to summarize data records (Fleming and Wallace, 1986; Smith, 1988). From a statistical perspective, the intention of summarizing a data record is to find a measure of central tendency what is representative for the data record. While the arithmetic mean is an appropriate method for many cases, there exist other means to summarize data records. Moreover, there exist a number of criteria for when to use which means to summarize a data record.
The first question when summarizing a data record is to ask what the data actually describe and what we intend to express with our summary. For a data record X, we can define a relationship we would like to conserve while replacing each single X_i with an average value $\bar{x}$. Based on this relationship, we can derive the appropriate method to summarize our record. For instance, our data record X describes the time elapsed for a test case and we want the keep the following relation, saying that the sum of all measurements $\sum_{i=1}^{n} X_i$ is equal to the total time elapsed T, defined as

$$T = \sum_{i=1}^{n} X_i = \sum_{i=1}^{n} \bar{x}.$$

Based on the term above, we can derive the definition of the method appropriate to summarize our data record with respect to the conserved relation what is the *arithmetic mean*, defined as

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} X_i. \tag{5.1}$$

Consider another example, similar to the one above, where the test case is a load test with a predefined number of users and the measurements $X = X_1, X_2, \ldots, X_{n-1}, X_n$ are measured as hits per second. Again, we have different measurements we want to summarize with respect to a relation to conserve. Each user drops the same number of requests T, whereby the hit rate X_i and the respective elapsed time $t_i = \frac{T}{X_i}$ vary. We

now conserve the relationship that the total number of requests for the data record is the sum of all hit rates X_i times the time elapsed t_i. Therefore, we want to substitute each hit rate X_i with an average value $\hat{x}$ so that the aforementioned relationship is conserved:

$$n \cdot T - \sum_{n}^{i=1} X_i t_i = 0 \Leftrightarrow n \cdot T - \hat{v} \sum_{n}^{i=1} t_i = 0 \Leftrightarrow n = \hat{x} \sum_{n}^{i=1} \frac{1}{X_i}$$

Similar to Eq. 5.1 we can derive the definition for the summarization method to use from the equation above what is the *harmonic mean*, defined as

$$\hat{x} = n \cdot \left(\sum_{i=1}^{n} \frac{1}{X_i} \right)^{-1} = \frac{n}{\frac{1}{X_1} + \frac{1}{X_2} + \ldots + \frac{1}{X_{n-1}} + \frac{1}{X_n}}. \tag{5.2}$$

We see that the summarization methods presented in Eq. 5.1 and 5.2 are useful for different types of data records, and should be used accordingly. The arithmetic mean is suitable for records, where the total sum of single measurements has a meaning, whereas the harmonic mean is suitable for measured rates or frequencies (Smith, 1988).
While the aforementioned measures of central tendency should be appropriate for most cases, they are not always the best choice though since the arithmetic and harmonic mean are heavily influenced by extreme observations (Shanmugam and Chattamvelli, 2015). For instance, for the data record $X = 1, 2, 3, 30$, three of four values are smaller than the arithmetic mean $\bar{x} = 9$. One option is to explicitly exclude outlier values from the data record. The so-called *trimmed mean* is obtained by truncating a upper and/or lower percentage t of the data record and, consequently, computing the (arithmetic or harmonic) mean for the remaining data record (Shanmugam and Chattamvelli, 2015). While this method is suitable to omit the effects of outliers, one still needs to specify which upper and/or lower percentage t needs to be truncated. Moreover, the use a (trimmed) mean requires the data records' frequencies to be distributed symmetrically around its mean. A probability distribution is skewed (and therefore asymmetric) if, graphically speaking, its histogram is not symmetric around its measure of central tendency. A simple method to measure the skewness of a probability distribution is *Bowley's measure* (Shanmugam and Chattamvelli, 2015), defined as

$$B_S = \frac{(Q_3 - M) - (M - Q_1)}{Q_3 - Q_1} = \frac{(Q_3 + Q_1 - 2M)}{Q_3 - Q_1}, \tag{5.3}$$

where Q_1 and Q_3 denote the first and third quartile, and M denotes the median of the probability distribution. The quartiles Q_i with $i \in \{1, 2, 3\}$ are defined as the values of a data record X, so that $\frac{i}{4}$ of the values of X are smaller than Q_i. The *median* is defined as Q_2, i.e., a value $M \in X$ so that half of the values in X are smaller than M. The median itself is a more robust measure of central tendency than the aforementioned ones since it is less influenced by outliers and can be used for both skewed and symmetric data records (Shanmugam and Chattamvelli, 2015). For a given ascendingly ordered data record $X = X_1, X_2, \ldots, X_{n-1}, X_n$, we can compute the median as follows:

$$\text{Median}(X) = \begin{cases} X_{\frac{n+1}{2}} & \text{if } n \text{ is odd} \\ \frac{1}{2}\left(X_{\frac{n}{2}} + X_{\frac{n}{2}+1}\right) & \text{if } n \text{ is even} \end{cases} \text{ with } X_i \leq X_j \text{ and } i < j \leq n \tag{5.4}$$

5.3.2 Measures of Dispersion

Last, we take a look at different measures of spread or dispersion. Most commonly used are the *variance* σ^2 and the *standard deviation* $\sigma = \sqrt{\sigma^2}$ defined along the mean μ of a probability distribution as

$$\sigma^2 = \frac{\sum_{i=1}^{n}(X_i - \mu)^2}{n} \tag{5.5}$$

Similar to the (arithmetic or harmonic) mean, the variance and the standard deviation are heavily influenced by extreme observations (Shanmugam and Chattamvelli, 2015) and not the best choice in all cases. Instead, two more robust measures are the *median absolute deviation* (MAD) and the *inter-quartile range* (IQR). The MAD is defined as the median of the absolute deviations from the probability distributions' median (Molyneaux, 2014), or, defined as follows:

$$\text{MAD}(X) = \text{Median}\Big(|X - \text{Median}(X)|\Big) \tag{5.6}$$

The IQR, however, defines the range between the first and the third quartile, Q_1 and Q_3 (Shanmugam and Chattamvelli, 2015). This range is larger for a widespread data record and smaller for a data range with a narrow spread, but is not influenced by extreme observations as those outliers do not lie within the range $[Q_1, Q_3]$. The IQR is defined as

$$\text{IQR}(X) = Q_3 - Q_1. \tag{5.7}$$

5.3.3 When to use which measure?

In this section we discussed the arithmetic and harmonic mean as well as the median as a measure of central tendency, the symmetric property that is required to use the arithmetic or harmonic mean, and different measures of spread for a given data record. At the beginning, we have raised the question of how to summarize data records (a) for an experiment repeated multiple times, and (b) obtained from different benchmarks. While for case (b) the answer is to use the arithmetic or harmonic mean (depending on the quality of the measurements), for (a) the answer is a little more elaborate.
The arithmetic (or harmonic) mean can be used whenever the quality of the measurement is appropriate and the data record is symmetric. According to Shanmugam and Chattamvelli (2015), the arithmetic mean is preferable "when the numbers combine additively to produce a resultant value", such as time periods or memory sizes, whereas the harmonic mean is preferable "when reciprocals of several non-zero numbers

combine additively to produce a resultant value", such as rates or frequencies (Smith, 1988). The median is a less precise estimator than the both means, yet more robust with regard to extreme observations. In addition, the MAD or IQR provide a more robust means of spread than the standard deviation.

5.4 Summarization Strategies

Besides the statistical considerations regarding the summarization of performance measurements, we are also challenged by the dimensionality of our performance evolution history. Given a record of performance measurements for a number of versions across different variants, we require statistical and numerical means to summarize performance evolution with single values. For this purpose, we propose two metrics addressing the following two aspects. First, different variants usually exhibit different levels of performance measurements, aside from evolution and fluctuations. To effectively compare the performance change of different variants, we propose to use the relative performance change as a suitable metric. Second, performance changes do not necessarily affect all variants in a similar way. To measure the homogeneity of performance changes across different variants, we propose to use the variance of changes per version. A mathematical definition for both relative performance change and performance change variance is given below.

5.4.1 Relative Performance Change

Let P be a $M \times N$ matrix with performance measurements $p_{i,j}$ with $i, j \in \mathbb{N}, i \leq M, j \leq N$ for M versions and N variants. Now we compute the relative performance change as a $M \times N$ matrix P' as follows:

$$p'_{i,j} = \begin{cases} 0 & \text{if } i = 1 \\ \frac{p_{i,j}}{p_{i-1,j}} - 1 & \text{else} \end{cases} \tag{5.8}$$

The relative performance change provides a normalized means to compare performance changes for different variants of a configurable software systems.

5.4.2 Performance Change Variance

Let P' be a $M \times N$ matrix with relative performance change measurements. To measure the homogeneity of performance changes across different variants, we compute v_i the variance all variants' relative performance changes for a version i with $i \leq M$ as follows:

$$v_i = \mathrm{Var}\{p'_{i,1}, p'_{i,2}, \ldots, p'_{i,N-1}, p'_{i,N}\} \tag{5.9}$$

The variance increases the more the relative performance changes spread, i.e., the more heterogeneously different variants evolve.

6 Evaluation

In the last three chapters, we have presented our methodology to assess the performance evolution history of configurable software systems. The first part covered a catalog of methods to derive a variability model from the software system or related resources along with different strategies to select sample sets of variants. In the second part, we presented and evaluated different strategies to select a subset of revisions, for which performance measurements approximate the overall performance evolution history. The third part reviewed aspects on concrete performance measurement, including guidelines to follow when choosing a benchmark to test, and when summarizing measurement results across different variants and versions.
While the evaluation of revision sampling strategies in the previous chapter already presented some performance measurement results ex ante, in this chapter, we evaluate the applicability of our methodology with a case study. With our case study we intend to answer the following research questions:

RQ_1) *Can we recover a performance evolution history?*

RQ_2) *Does performance evolve for configurable software systems?*

RQ_3) *What revision sampling strategies accurately sketch performance evolution history?*

RQ_4) *Following our methodology, do we obtain reliable performance measurements?*

This chapter is organized as follows. In section 6.1 we present our case study corpus, i.e., the selection of configurable software systems that we assess performance for in our experiment. Section 6.2 addresses RQ_1 by documenting how we followed our methodology throughout the experiment setup and conduction. Section 6.3 addresses RQ_2 with an extensive description of performance evolution results and insights obtained from it. Finally, section 6.4 addresses RQ_3 and RQ_4 by reviewing the accuracy of revision sampling strategies and assessing the reliability of our performance measurement approach.

6.1 Case Study Corpus

To evaluate our methodology, we selected two subject systems: GNU XZ and x264. The selection process accounted for the following requirements. First, since our intention is to assess the performance evolution history of a configurable software system, we limited our selection to mature software systems that exhibit a development history of a couple years or more. Second, we limit our selection to software systems for which we can obtain a fine-grained development history, usually version control logs. The rationale is that our methodology considers the sampling of different revisions as well

as the possibility to manually inquire possible causes of performance changes. Third, we intend to consider software systems that have already been subject of related research, such as work on performance prediction models. This enables us to compare our results with, and place our results in the context of previous work. Lastly, the scope of this case study is constrained by limited time. Hence, the selection of software systems to assess is not representative, yet intended to validate the methodology presented earlier, and to obtain empirical insights on whether, and if so, how performance evolves for configurable software systems.

Our case study corpus comprises two configurable software systems, GNU XZ[1] and x264[2]. GNU XZ is a free file compression tool that is widely used across the open-source universe. GNU XZ provides a development history of about ten years, or more than 1,100 versions, and is actively maintained as part of the GNU project. It provides a publicly accessible Git repository as well as additional resources, such as archived mailing lists and bug reports. The software itself has not been subject of related research, yet it has been frequently compared with other file compression tools with respect to compression performance.

x264 is a free library implementation of the H.264 codec for video encoding, commonly known as MPEG-4. The implementation provides a command line interface to use and has been subject of previous research, including performance prediction models (Siegmund et al., 2012, 2015). Similar to GNU XZ, x264 is a mature software system that provides a development history of almost ten years, or more than 2,800 versions. Both software systems are configurable at load-time via command line arguments that modify the file compression process, or video encoding process, respectively.

6.2 RQ_1: Can we recover a performance evolution history?

In the following, we document the application of our methodology to the two software systems of our case study corpus. In particular, this section covers the aspects of variability- and performance assessment since we have already covered the results for revision sampling in chapter 4.

All performance measurements made in the following were conducted on a Linux machine (Debian 8) with a Intel Xeon E5-2680 v2 with 2.8 GHz and 16 GB RAM. For each version and variant, we repeatedly executed a benchmark five times and selected the median of the resulting measurements to minimize the impact of biased measurements.

6.2.1 Application of our methodology to GNU XZ

Variability Assessment. For GNU XZ, we considered the application's man page documentation to synthesize a variability model since we could not find any more comprehensive documentation artifacts (cf. the use cases in Table 3.1 as well as the questionnaire in Table 3.2). GNU XZ is configurable at build-time via command-line arguments passed to the program with every execution. Since the application is a file

[1] Find the project description of GNU XZ at `https://tukaani.org/xz/`.

[2] Find the project description of x264 at `https://www.videolan.org/developers/x264.html`.

compression utility, it provides two operation modes, file compression and decompression. For our evaluation, we chose to assess performance for file compression as most compression tools are compared by compression performance rather than decompression. We extracted configuration options based on two criteria from the man page documentation. First, a command line argument is a valid configuration option if it relates to the chosen operation mode. This specifically excludes options, such as command line arguments to return the version number, or to affect user interaction. Second, a command line argument is a valid configuration option if it does not alter the configuration of other options. This especially applies to presets which on the one hand relate to the chosen operation mode, but on the other hand preselect values for other configuration options. While for some configuration parameters, domains were explicitly specified, for those options remaining unspecified we had to manually investigate minimum or maximum values by trial-and-error. In total, we identified nine binary, four numeric configuration options, and two constraints, resulting in 7.22×10^{14} possible variants considering the domains of numeric configuration options.
Using pair-wise sampling (cf. Section 3.3), we selected 36 variants for which we assess their performance. We selected pair-wise sampling over the other mentioned sampling methods as we cover all pair-wise feature interactions with very few configurations. The feature model in terms of configuration option parameters has not changed during the development history as the man page has never been revised significantly. In fact, the documentation lists a number of configuration options whose corresponding features have not been implemented yet, for instance support for multi-threading.

Performance Assessment. For GNU XZ, we selected the Canterbury corpus (2.8 MB) as the benchmark. The Canterbury corpus is a standardized set of files that is intended to be representative since it contains files of different types, such as text or binary data. To measure performance of GNU XZ, we refer to the execution time since the benchmark file as well as the implemented compression algorithm remain unchanged for all versions and variants. We conceive any increase in execution time as performance degradation, and any decrease in execution time measurements as an increase in performance quality properties. Further possible performance indicators for GNU XZ beside the execution time include the compression ratio (ratio of uncompressed input and compressed output), and the resource utilization. With respect to the limited scope of the thesis, and, for the sake of comparability, we only take into account execution time.

6.2.2 Application of our methodology to x264

Variability Assessment. For x264, similar to GNU XZ, the most comprehensive documentation, and, therefore our primary source of information for the synthesis of a variability model was the software system's man page. x264 is configurable at load-time and can be configured via command-line arguments passed when executing the program. The software provides only one operation mode, the encoding of uncompressed video data, yet it can be tuned with a range of command-line parameters. We only selected those parameters as configuration options that relate to the operation mode, and excluded preset parameters. For instance, we excluded a command-line parameter that specifies how many video frames at the beginning of the video should

be dropped. In total, we identified eight binary options, twelve numeric options, and no constraint, resulting in 7.98×10^{23} variants considering the domains of numeric configuration options.
Using pair-wise sampling (cf. Section 3.3) we selected a sample set of eight variants for which we assess their performance. As the binary configuration options are optional, we required very few configurations to cover all pair-wise feature interactions.

Performance Assessment. We selected an uncompressed video file (79.9 MB) provided by the Xiph.org foundation as a benchmark to encode as an MP4 file. Similar to GNU XZ, we consider the execution time as the key performance indicator since the benchmark file as well as the implemented encoding standard remain unchanged for all versions and variants. We conceive any increase in execution time as performance degradation, and any decrease in execution time measurements as an increase in performance quality properties. Again, we could consider additional performance indicators, such as throughput in terms of frames encoded per second. However, with respect to the limited scope of this thesis, and, to compare performance results across systems, we decided to only focus on execution time.

All in all, the methodology provided a guideline to obtain performance evolution results. In particular, for the synthesis of variability models as well as the selection of suitable performance benchmarks, the methodology provided conceptual decisions in terms of which use case to refer to. However, the methodology remains generic rather than concrete in other aspects, such as the recommendation of variant- or version sampling strategies. For variant sampling, the specification of coverage criteria is left to the practicioner, and, for version sampling, the suitability of a sampling strategy might also depend on the shape of the overall performance evolution history (cf. section 4.4).

6.3 RQ_2: Does performance evolve for configurable software systems?

We have evaluated the performance evolution history results regarding two aspects, effect magnitude and effect range. For both aspects, we ask whether there are patterns or trends. A pattern in our context is a recurring shape of segments of an performance history curve. In addition, we define a trend as a more global increase or decrease in performance measurements that might superpose local patterns.

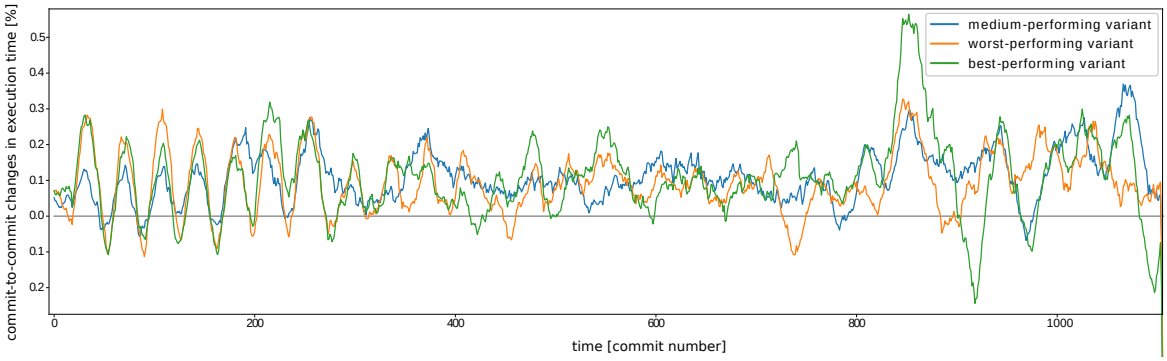
6.3.1 Effect magnitude

To investigate the magnitude of changes in performance measurements, we consider the relative changes from commit to commit rather than the absolute changes since most variants exhibited different levels of performance measures as some variants performed better than others, depending on the respective configuration (cf. section 5.4.1). To illustrate the relative performance changes, in Figure 6.1 we present the relative performance changes of the best- and worst-performing variants along with a medium performing variant for both systems respectively. In addition, we provide a zero base-

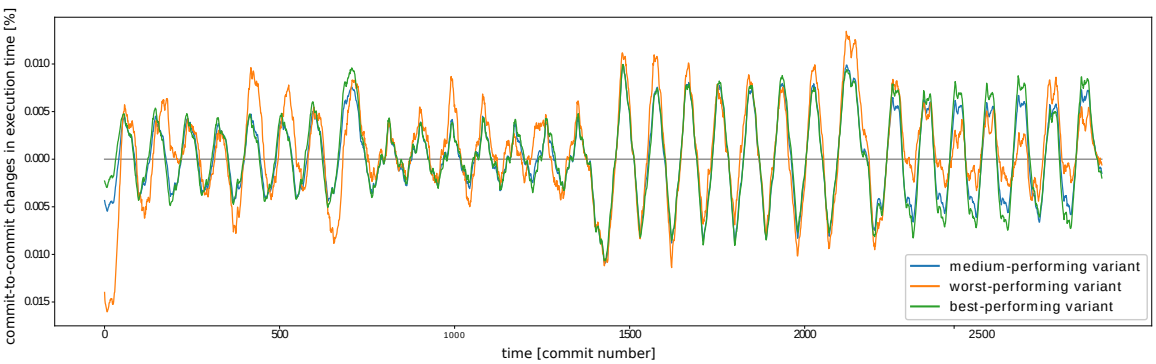
line (grey) to illustrate whether performance measurements increase (curve above the baseline) or decrease (curve below the baseline).

Patterns. For both systems we can see local fluctuations in the relative performance changes. For GNU XZ, however, the different versions fluctuate more heterogeneously starting from commit 250.
For this first commit segment as well as for x264, commits with a performance degradation have commit messages indicating the introduction of new features to the software system. For commits for which performance improves, commit messages suggest that program errors have been fixed.

Trends. With respect to our zero-change baseline, we can see that for GNU XZ, the vast majority of commits result in a performance regression, while for x264, the distribution of performance-increasing and -decreasing commits is more balanced. This indicates a global trend for GNU XZ, where performance has degraded over the course of ten years, while for x264, performance, besides more local fluctuations, has not degraded significantly.



(a) GNU XZ



(b) x264

Fig. 6.1. Relative commit-to-commit change in execution time in percent for GNU XZ and x264. Depicted are curves for three variants for both systems respectively. The illustrated best-, medium-, and worst-performing variant exhibited the best, average, and worst average execution time across all versions.

6.3.2 Effect range

To summarize the performance evolution with respect to the effect range, we ask the question whether multiple variants change homogeneously for the same version. Therefore, we first computed the relative performance change form commit to commit for each variant. Second, we have computed the variance of relative performance changes per version. The rationale behind this is that if for a new version all variants' performance changes homogeneously, the resulting variance is low, whereas if variants change heterogeneously, the resulting variance is high (cf. section 5.4.2). For both GNU XZ and x264, we present the variance of performance changes over time in Figure 6.2. In addition to the smoothed curve, we provide the global average variance to indicate global trends.

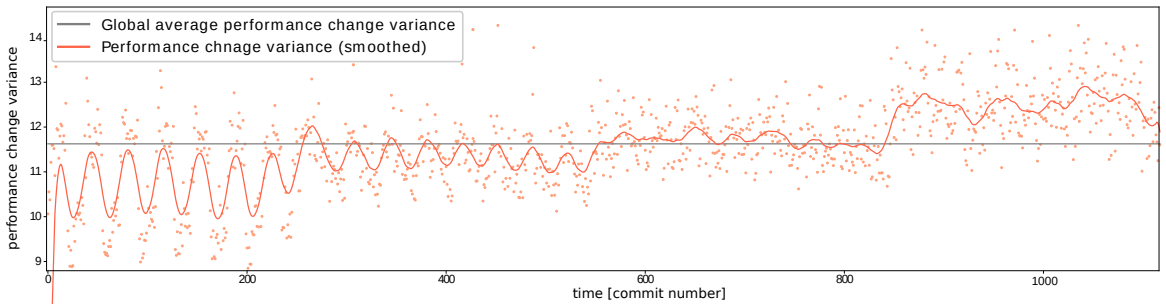
Patterns. Similar to the effect magnitude, we can see local fluctuations in the variance of performance changes for the beginning of GNU XZ's history as well as throughout all of x264's history. These fluctuations suggest that for some time, variants evolved more homogeneously, followed by a time span where variants evolved more independently.

Trends. For GNU XZ, we identified a global increase in variance among performance changes while simultaneously, the amplitude of fluctuations decreased. That is, variants continued evolving more heterogeneously over the development history. Moreover, the older the software system, the fewer versions seem to change performance of all variants. For x264, however, we observe more fluctuations throughout all the history, indicating that from time to time commits did affect more variants than others. In addition, the variance does not increase on a global scope. This indicates that most variants kept evolving homogeneously throughout the development history.

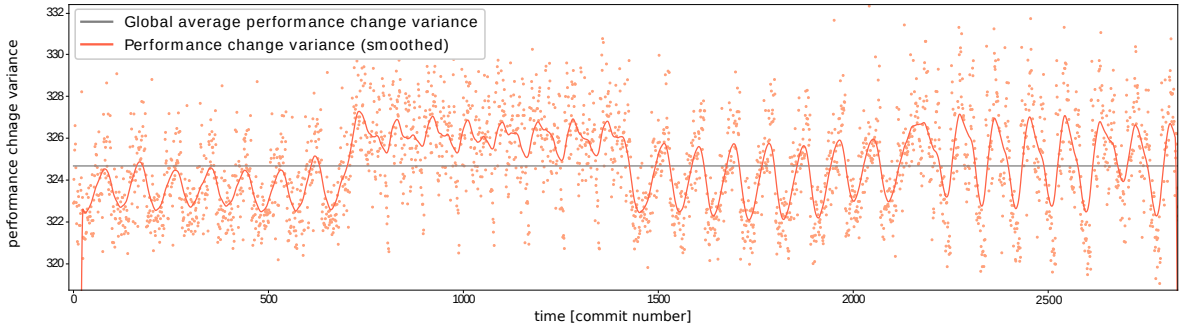
6.3.3 What can be learn from a performance history?

The described performance results in the previous section suggest that both software systems exhibit different quality attributes with respect to their software architecture. In this last subsection, we place the observed results in the context of existing work on software evolution mentioned in section 2.2.

GNU XZ. For GNU XZ, based on our observations, we can say that the software architecture in the beginning was ductile in the beginning and got more brittle over the time since fluctuations in the performance change variance decreased; also, the variance of performance changes increased during the development history suggesting a more chaotic performance evolution for each variant. This is in line with our observation in Figure 4.4, where the best-performing variant changes significantly stronger compared to the other two variants. The concept of technical debt (Guo et al., 2011), meaning a global trend of performance degradation can be identified as well. Interestingly, GNU XZ was revised more frequently in the beginning of the development history than more recently. All in all, the observations suggest that GNU XZ today exhibits a poor software architecture that has become brittle as most of the committed versions are older than five years and very few commits did affect variants homogeneously.



(a) GNU XZ



(b) x264

Fig. 6.2. Variance of relative execution time changes across all variants of a software system. While the markers depict values for a single version/commit, the smoothed curve visualizes local trends. The baseline depicts the global average variance of relative performance change.

x264. In contrast to GNU XZ, the observations for x264 indicate that the software architecture remained ductile during the development history for a number of reasons. First, the fluctuations in the variance of performance changes did not decrease significantly over time indicating that more recent commits can have the same effect on performance as more older ones. Moreover, the overall variance of performance change, besides the aforementioned fluctuations, did not increase. Second, the performance measurements for x264 fluctuate, but do not exhibit a global trend similar to GNU XZ. This shows that both the effect magnitude as well as the effect range for performance changes remained stable during the development history. All in all, the observations suggest that the software architecture of x264 is more mature than GNU XZ.

6.4 RQ_3 and RQ_4: Accuracy and Reliability

In this final evaluation section we evaluate the methodology with respect to accuracy (RQ_3) and (RQ_4). RQ_3 asks, whether our methodology can accurately describe performance evolution for a given configurable software system, and RQ_4 asks, whether the measurements we obtain our performance evolution history from are reliable.

Accuracy. With regard to the partial evaluation of revision sampling strategies in chapter 4, where we evaluated accuracy, no definitive answer can be given. The used

case study of only two configurable software systems is too small to make a reliable statement here. However, what we have learned from the two software systems was that we could not clearly distinguish accuracy measures for different revision sampling strategies for x264, presumably due to a performance evolution history curve that only exhibits small-sized fluctuations. In other words, the curve for x264 is shaped quasi-linear and therefore easily approximated by any selection of sample points. Leaving the indecisive accuracy evaluation for x264 as well as the size of the case study corpus aside, at least, the accuracy results for GNU XZ suggest that the best-performing revision sampling strategies can accurately select a sample of revisions to sketch performance evolution. Nonetheless, this question requires further research to reliably recommend and ensure a accurate revision sampling strategy.

Reliability. To assess reliability for our methodology, we ask whether performance measurements using our methodology are reproducible, i.e., if we obtain similar measurements under similar conditions. We answer RQ_4 by conducting a small case study. We measure the spread of execution time measurements to check whether spread is influenced by the software system tested, the execution time, or the number of repetitions. During all experiments in our evaluation, we have used five repetitions per version and variant. This experiment is merely intended to validate the reliability of this decision.

For this experiment, of the two sample system GNU XZ and x264, we selected three variants each. Each of the three variants is derived from configuration presets with the incentive to obtain a fast-, medium-, and a slow-performing variant. The rationale behind this approach is that conclusions about a configurable software system drawn from multiple variants are more valid than simply measuring one arbitrary variant. In addition, this choice allows us to investigate the influence of the execution time on spread as we expect different levels of execution time measurements. Each experiment was repeated three to 20 times. Finally, since the execution time measurements for the Canterbury corpus for GNU XZ were relatively small (around one second), we tested the same three variants with an additional benchmark which is the Canterbury corpus tripled in size.

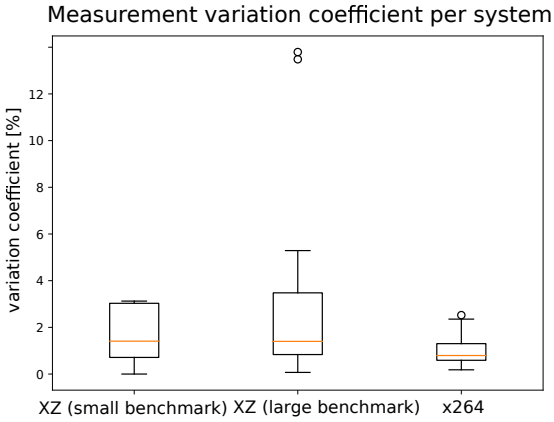


Fig. 6.3. Distributions of variation coefficients for GNU XZ (for two different benchmarks), and for x264. Measured median variation coefficients were 1.408 % and 1.40 % for GNU XZ, and 0.794 % for x264.

First, we investigated the influence of the configurable software system studied. Therefore, in Figure 6.3, we illustrate the distribution of the variation coefficients for each experiment per system. The variation coefficient is computed as the standard deviation of the execution time divided by (or normalized to) the arithmetic mean in order to compare the measures of spread for arbitrary distributions. We see that for GNU

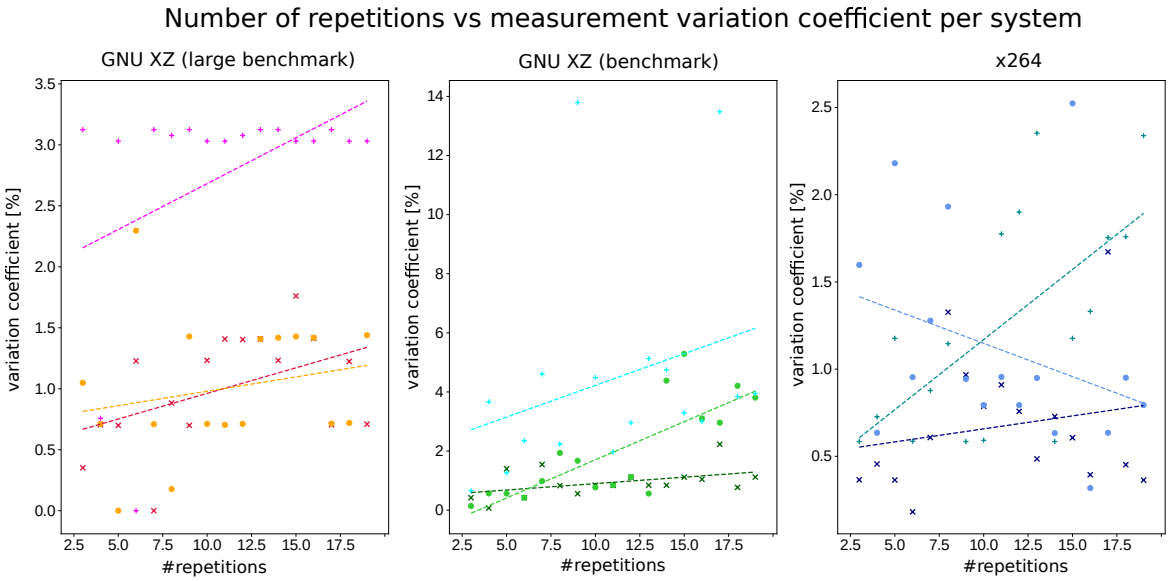


Fig. 6.4. Number of repetitions vs execution time measurement variation coefficients for GNU XZ and x264. For GNU XZ, we tested two benchmarks differing in size (the Canterbury corpus and a triple copy thereof). For each subplot, we present three variants, one slow-, one medium-, and one fast-performing one, each depicted by different colors and markers (`x`: slow, `o`:medium, `+`: fast).

XZ, regardless of the benchmark, the variation coefficient is almost two times higher than for x264. This observation indicates, that for GNU XZ, the execution time measurement variance is notably higher, suggesting that the software system tested has an influence on how strongly measurements can spread.
Second, we investigated the influence of the number of repetitions on the variation coefficient of execution time measurements. In Figure 6.4, we illustrate the variation coefficients plotted against the number of repetitions. We colorized each variant uniquely, and assigned markers for each variant (`x` for the slow-, `o` for the medium-, and `+` for the fast-performing variants). In addition, we provided best linear fit curves of the same color to identify possible trends. However, non of the presented fits suggests a significant relationship.
In conclusion, the only significant influence we could determine in our small reliability assessment was the influence of the configurable software system studied. The results suggest, that the software system studied influences the reliability of execution time measurements. Further investigation, in particular with regard to the question, what properties of a configurable software system drive the impact on reliability, is required to determine the degree of (non-)determinism of a configurable software system. However, this question exceeds the scope of this thesis. All in all, we conclude that performance, or execution time measurements using the Unix `time` command are sufficiently reliable for the purpose of assessing the performance evolution of configurable software systems.

7 Conclusion

Next, we conclude this thesis, review the limitations of this work, and provide an outlook for future work on the topic of performance evolution of configurable software systems.

7.1 Concluding Remarks

In this thesis, we have presented a methodology to assess the performance evolution history of configurable software systems. Our methodology provides a comprehensive and informed means to analyze performance of software systems with respect to the dimensions of variability and evolution.

Methodology summary. In the first part of our methodology, chapter 3, we provide an user guideline about how to synthesize a variability model describing valid selections of configuration options. The guideline is driven by the extent of which variability is documented for the software system; for three different scenarios (cf. Table 3.1), we advocate the use of existing variability models, family-based analysis approaches, or a bottom-up strategy to pursue based on documentation assets (cf. Table 3.2). The second part of our methodology in chapter 4, has addressed the question of how to select a subset of versions of a configurable software system to sketch performance evolution accurately and efficiently. We have proposed and evaluated five different strategies based on assumed relationships between software metrics and the impact of a revision on the software system's performance. The findings of our evaluation suggest that versions, for which large code sections are revised, are a good approximation for such version sample sets. Moreover, we were able to learn from larger revisions the possible impact of modifying a file on the overall software system performance. We have successfully tested the sampling strategies on a mature configurable software system with a development history of about ten years. Finally, in the third part of our methodology in chapter 5, we have presented the practical aspects of performance measurement, including the selection of suitable performance benchmarks, the choice of profiling tools, and statistical means to summarize measurement results. In addition, we present statistical means to summarize and compare performance measurements across variants.
We have evaluated our methodology with a case study of two configurable software systems (GNU XZ and x264). Using our methodology, for both software systems, we were able to obtain a performance evolution history of around a decade each. We identified common patterns, among others a direct relationship between performance degradation (execution time measurements increase) and revisions indicating the introduction of new functionality. In addition, we observed performance improvements for revisions indicating bug-fixes and refactorings. However, both software systems exhibited

different global trends regarding their performance evolution history which suggests different levels of maturity. As a more mature configurable software system, for x264, all tested variants evolved homogeneously, while for GNU XZ, most tested variants evolved heterogeneously and independent from each other. In fact, a subsequent reliability assessment of the performance measurement tool indicated that measurement spread was merely dependent on the software system tested, and was lower for the more mature software system x264.

Research Contributions and Limitations. We contribute a methodological framework to obtain performance measurements for variant- and version-rich software systems. This way, provide practitioners and the research community an integrated and comprehensive set of guidelines for analyzing and understand a configurable software system's performance in two dimensions, time and variability. Besides the methodological guidelines, we provide the implementation of our experiment setup, used sampling strategies, and performance measurement results for GNU XZ and x264 at `http://www.github.com/smba/SPLPioneerPublic` to enable further research and work in this field. As a further substantial contribution, we have proposed and evaluated four revision sampling strategies to assess performance as close as possible to the whole population of the performance of all revisions. As we have learned from the evaluation of version sampling strategies as well as the overall evaluation, the software system tested can have a substantial impact on the methodology accuracy. Therefore, we push for more research that is still required to render precisely the impact for software systems of different levels of maturity and from different domains.

7.2 Outlook and Future Work

In the course of this thesis, several aspects arose that can be taken into account for future work on the assessment of performance evolution for configurable software systems. For the revision sampling strategy changed-files sampling (cf. section 4.2.5), we have learned and estimated the impact of modifying a single file on the software system's performance. Regarding this sampling strategy driven by machine-learning, we propose the following possible extensions that we believe sketch possible future directions. First, the current features mapped to the performance influence are files. While the sampling strategy with this level of granularity can be easily applied to arbitrary software systems, more fine-grained feature-to-impact mappings are possible. For instance, instead of files, functions or methods could be conceived as features to map to performance impact estimates. Although this requires additional language-specific parsing, we believe this to be a promising extension that might further increase the accuracy of this sampling strategy. Second, the learned knowledge about the impact of modifying a specific file (or function) can be used to localize those code sections that are likely to impact performance. This knowledge, for instance, can be used to advise future developers to be aware of the possible impact of modifying a certain file. This might sketch a good basis for possible extensions to integrated development environments.

Bibliography

Al-Hajjaji, M., Krieter, S., Thüm, T., Lochau, M., and Saake, G. (2016). IncLing: efficient product-line testing using incremental pairwise sampling. pages 144–155. ACM Press.
Cited on page 28.

Al-Kofahi, J., Nguyen, T., and Kästner, C. (2016). Escaping AutoHell: a vision for automated analysis and migration of autotools build systems. In *Proceedings of the 4th International Workshop on Release Engineering*, pages 12–15. ACM.
Cited on page 19.

Alves, V., Schwanninger, C., Barbosa, L., Rashid, A., Sawyer, P., Rayson, P., Pohl, C., and Rummler, A. (2008). An exploratory study of information retrieval techniques in domain analysis. In *Software Product Line Conference, 2008. SPLC'08. 12th International*, pages 67–76. IEEE.
Cited on pages 4, 19, 20, and 22.

Andersen, N., Czarnecki, K., She, S., and Wąsowski, A. (2012). Efficient synthesis of feature models. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 106–115. ACM.
Cited on page 4.

Antony, J. (2014). Design of Experiments and its Applications in the Service Industry. In *Design of Experiments for Engineers and Scientists*, pages 189–199. Elsevier. DOI: 10.1016/B978-0-08-099417-8.00010-9.
Cited on pages 28 and 29.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines*. Springer Berlin Heidelberg, Berlin, Heidelberg. DOI: 10.1007/978-3-642-37521-7.
Cited on pages 1, 2, 7, 13, 17, 28, and 29.

Bakar, N. H., Kasirun, Z. M., and Salleh, N. (2015). Feature extraction approaches from natural language requirements for reuse in software product lines: A systematic literature review. *Journal of Systems and Software*, 106:132–149.
Cited on pages 4, 19, 20, and 22.

Batory, D. (2005). Feature models, grammars, and propositional formulas. In *SPLC*, volume 3714, pages 7–20. Springer.
Cited on pages 7, 25, and 26.

Benavides, D., Segura, S., Trinidad, P., and Cortés, A. R. (2007). FAMA: Tooling a Framework for the Automated Analysis of Feature Models. *VaMoS*, 2007:01.
Cited on page 25.

Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2005a). Automated Reasoning on Feature Models. *CAiSE*, 5(3520):491 – 503.
Cited on pages 25 and 26.

Benavides, D., Trinidad, P., and Ruiz-Cortés, A. (2005b). Using Constraint Programming to Reason on Feature Models. *SEKE*, pages 677 – 682.
Cited on page 25.

Breivold, H. P., Crnkovic, I., and Larsson, M. (2012). A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16–40.
Cited on pages 8 and 9.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley Publishing Co.
Cited on page 7.

Darringer, J. A. and King, J. C. (1978). Applications of symbolic execution to program testing. *Computer*, 11(4):51–60.
Cited on page 18.

Dietrich, C., Tartler, R., Schröder-Preikschat, W., and Lohmann, D. (2012). A robust approach for variability extraction from the Linux build system. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 21–30. ACM.
Cited on page 1.

Fleming, P. J. and Wallace, J. J. (1986). How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221.
Cited on page 53.

Foo, K. C., Jiang, Z. M., Adams, B., Hassan, A. E., Zou, Y., and Flora, P. (2010). Mining performance regression testing repositories for automated performance analysis. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 32–41. IEEE.
Cited on page 51.

German, D. M. and Hindle, A. (2006). Visualizing the evolution of software using softChange. *International Journal of Software Engineering and Knowledge Engineering*, 16(01):5–21.
Cited on page 15.

Gosain, A. and Sharma, G. (2015). A Survey of Dynamic Program Analysis Techniques and Tools. In Satapathy, S. C., Biswal, B. N., Udgata, S. K., and Mandal, J., editors, *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*, volume 327, pages 113–122. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-11933-5_13.
Cited on page 52.

Guo, J., Czarnecki, K., Apely, S., Siegmundy, N., and Wasowski, A. (2013). Variability-aware performance prediction: A statistical learning approach. In *Proceedings of*

the 28th IEEE/ACM International Conference on Automated Software Engineering, pages 301–311. IEEE Press.
Cited on pages 3, 13, 14, and 29.

Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F. Q. B., Santos, A. L. M., and Siebra, C. (2011). Tracking technical debt — An exploratory case study. pages 528–531. IEEE.
Cited on pages 3, 9, 41, and 62.

Haslinger, E. N., Lopez-Herrejon, R. E., and Egyed, A. (2011). Reverse Engineering Feature Models from Programs' Feature Sets. pages 308–312. IEEE.
Cited on pages 21 and 23.

Haslinger, E. N., Lopez-Herrejon, R. E., and Egyed, A. (2013). On Extracting Feature Models from Sets of Valid Feature Combinations. In *FASE*, volume 13, pages 53–67. Springer.
Cited on pages 21 and 23.

Heger, C., Happe, J., and Farahbod, R. (2013). Automated root cause isolation of performance regressions during software development. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, pages 27–38. ACM.
Cited on pages 3, 12, 31, 34, 37, and 51.

Hindle, A. (2015). Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, 20(2):374–409.
Cited on page 15.

Huang, A. (2008). Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, pages 49–56.
Cited on page 34.

Hunsen, C., Zhang, B., Siegmund, J., Kästner, C., Leßenich, O., Becker, M., and Apel, S. (2016). Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, 21(2):449–482.
Cited on page 1.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst.
Cited on pages 7 and 21.

King, J. C. (1976). Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.
Cited on page 18.

Kästner, C., Apel, S., and Kuhlemann, M. (2009). A model of refactoring physically and virtually separated features. In *ACM Sigplan Notices*, volume 45, pages 157–166.

ACM.
Cited on page 17.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *ACM SIGPLAN Notices*, volume 46, pages 805–824. ACM.
Cited on pages 17 and 18.

Liggesmeyer, P. (2009). *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Springer Science & Business Media.
Cited on pages 8 and 10.

Lillack, M., Kästner, C., and Bodden, E. (2014). Tracking load-time configuration options. pages 445–456. ACM Press.
Cited on page 18.

Linsbauer, L., Lopez-Herrejon, R. E., and Egyed, A. (2014). Feature Model Synthesis with Genetic Programming. In Le Goues, C. and Yoo, S., editors, *Search-Based Software Engineering: 6th International Symposium, SSBSE 2014, Fortaleza, Brazil, August 26-29, 2014. Proceedings*, pages 153–167. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-09940-8_11.
Cited on pages 20, 21, 22, and 23.

Lopez-Herrejon, R., Galindo, J., Benavides, D., Segura, S., and Egyed, A. (2012). Reverse engineering feature models with evolutionary algorithms: An exploratory study. *Search Based Software Engineering*, pages 168–182.
Cited on pages 20, 21, 22, and 23.

Lopez-Herrejon, R. E., Linsbauer, L., Galindo, J. A., Parejo, J. A., Benavides, D., Segura, S., and Egyed, A. (2015). An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103:353–369.
Cited on pages 20, 21, and 22.

Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., and Apel, S. (2016). A comparison of 10 sampling algorithms for configurable systems. In *Proceedings of the 38th International Conference on Software Engineering*, pages 643–654. ACM.
Cited on pages 28 and 29.

Medeiros, F., Ribeiro, M., Gheyi, R., Apel, S., Kastner, C., Ferreira, B., Carvalho, L., and Fonseca, B. (2017). Discipline matters: Refactoring of preprocessor directives in the# ifdef hell. *IEEE Transactions on Software Engineering*.
Cited on page 18.

Moin, A. and Khansari, M. (2010). Bug localization using revision log analysis and open bug repository text categorization. *Open Source Software: New Horizons*, pages 188–199.
Cited on page 31.

Molyneaux, I. (2014). *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 2 edition.
Cited on pages 2, 3, 10, 11, 12, 34, and 55.

Nadi, S., Berger, T., Kästner, C., and Czarnecki, K. (2014). Mining configuration constraints: Static analyses and empirical results. In *Proceedings of the 36th International Conference on Software Engineering*, pages 140–151. ACM.
Cited on pages 18, 19, 20, 21, and 22.

Nadi, S., Berger, T., Kästner, C., and Czarnecki, K. (2015). Where do configuration constraints stem from? an extraction approach and an empirical study. *IEEE Transactions on Software Engineering*, 41(8):820–841.
Cited on pages 4, 18, 19, 20, 21, 22, and 23.

Nguyen, H. V., Kästner, C., and Nguyen, T. N. (2014a). Building call graphs for embedded client-side code in dynamic web applications. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 518–529. ACM.
Cited on page 18.

Nguyen, H. V., Nguyen, H. A., Nguyen, T. T., and Nguyen, T. N. (2011). Auto-locating and fix-propagating for HTML validation errors to PHP server-side code. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pages 13–22. IEEE Computer Society.
Cited on page 18.

Nguyen, T. H., Nagappan, M., Hassan, A. E., Nasser, M., and Flora, P. (2014b). An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 232–241. ACM.
Cited on pages 3 and 12.

Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press.
Cited on pages 8 and 9.

Passos, L., Czarnecki, K., and Wąsowski, A. (2012). Towards a catalog of variability evolution patterns: the Linux kernel case. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, pages 62–69. ACM.
Cited on pages 2 and 10.

Passos, L., Padilla, J., Berger, T., Apel, S., Czarnecki, K., and Valente, M. T. (2015). Feature scattering in the large: a longitudinal study of Linux kernel device drivers. pages 81–92. ACM Press.
Cited on pages 2 and 9.

Peng, X., Yu, Y., and Zhao, W. (2011). Analyzing evolution of variability in a software product line: From contexts and requirements to features. *Information and Software Technology*, 53(7):707–721.
Cited on pages 2, 4, 9, and 10.

Perry, D. E. and Wolf, A. L. (1991). Software architecture. *Submitted for publication.*
Cited on pages 2 and 9.

Rabkin, A. and Katz, R. (2011). Static extraction of program configuration options. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 131–140. ACM.
Cited on pages 4, 19, 20, 22, and 23.

Sarkar, A., Guo, J., Siegmund, N., Apel, S., and Czarnecki, K. (2015). Cost-efficient sampling for performance prediction of configurable systems (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 342–352. IEEE.
Cited on pages 3, 14, 29, and 30.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-oriented programming of software product lines. *Software Product Lines: Going Beyond*, pages 77–91.
Cited on page 1.

Seidl, C., Heidenreich, F., and Aßmann, U. (2012). Co-evolution of models and feature mapping in software product lines. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 76–85. ACM.
Cited on pages 2 and 10.

Shanmugam, R. and Chattamvelli, R. (2015). *Statistics for scientists and engineers*. Wiley, Hoboken, New Jersey.
Cited on pages 54 and 55.

She, S., Lotufo, R., Berger, T., Wasowski, A., and Czarnecki, K. (2011). Reverse engineering feature models. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 461–470. IEEE.
Cited on pages 4, 22, and 26.

Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. (2015). Performance-influence models for highly configurable systems. pages 284–294. ACM Press.
Cited on pages 1, 3, 4, 13, 14, 28, 29, 30, and 58.

Siegmund, N., Kolesnikov, S. S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., and Saake, G. (2012). Predicting performance via automated feature-interaction detection. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 167–177. IEEE.
Cited on pages 3, 4, 13, 14, 28, 29, and 58.

Smith, J. E. (1988). Characterizing computer performance with a single number. *Communications of the ACM*, 31(10):1202–1206.
Cited on pages 53, 54, and 56.

Storey, M.-A. D., Čubranić, D., and German, D. M. (2005). On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202. ACM.
Cited on page 15.

Tamrawi, A., Nguyen, H. A., Nguyen, H. V., and Nguyen, T. N. (2012). Build code analysis with symbolic evaluation. In *Proceedings of the 34th International Conference on Software Engineering*, pages 650–660. IEEE Press.
Cited on page 19.

Tartler, R., Dietrich, C., Sincero, J., Schröder-Preikschat, W., and Lohmann, D. (2014). Static Analysis of Variability in System Software: The 90, 000# ifdefs Issue. In *USENIX Annual Technical Conference*, pages 421–432.
Cited on pages 28 and 29.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014). A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys*, 47(1):1–45.
Cited on pages 1 and 17.

Thüm, T., Batory, D., and Kastner, C. (2009). Reasoning About Edits to Feature Models. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 254–264, Washington, DC, USA. IEEE Computer Society.
Cited on page 7.

White, J., Dougherty, B., and Schmidt, D. C. (2009). Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284.
Cited on page 1.

Williams, A. W. and Probert, R. L. (1996). A practical strategy for testing pair-wise coverage of network interfaces. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 246–254. IEEE.
Cited on pages 28 and 29.

Woodside, M., Franks, G., and Petriu, D. C. (2007). The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE'07*, pages 171–187. IEEE.
Cited on pages 3, 11, and 12.

Wu, J., Holt, R. C., and Hassan, A. E. (2004). Exploring software evolution using spectrographs. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 80–89. IEEE.
Cited on page 15.

Xu, T., Jin, L., Fan, X., Zhou, Y., Pasupathy, S., and Talwadker, R. (2015). Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. pages 307–319. ACM Press.
Cited on page 19.

Zhang, B., Becker, M., Patzke, T., Sierszecki, K., and Savolainen, J. E. (2013). Variability evolution and erosion in industrial product lines: a case study. In *Proceedings of the 17th International Software Product Line Conference*, pages 168–177. ACM.
Cited on pages 2 and 9.

Zhang, Y., Guo, J., Blais, E., and Czarnecki, K. (2015). Performance prediction of configurable software systems by fourier learning (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 365–373. IEEE.
Cited on pages 4, 13, and 14.

Zhou, S., Al-Kofahi, J., Nguyen, T. N., Kästner, C., and Nadi, S. (2015). Extracting configuration knowledge from build files with symbolic analysis. In *Proceedings of the Third International Workshop on Release Engineering*, pages 20–23. IEEE Press.
Cited on pages 4 and 19.